Proceedings of the 2010

IEEE International Symposium on Workload Characterization (IISWC'10)

December 2 - 4, 2010

Atlanta, GA, USA

IISWC 2010
IEEE International Symposium on Workload Characterization

IEEE Catalog Number: CFP10236-PRT

ISBN: 978-1-4244-9295-4

Message from the General Chair

It is my great pleasure to welcome you all to Atlanta, Georgia for the 2010 IEEE International Symposium on Workload Characterization (IISWC-2010). This is the first time IISWC being held in the Southeast region in the States. The symposium will take place in the heart of the city at Georgia Tech Hotel located in-between the downtown and midtown areas. I wish you enjoy the richness of the surroundings offered by the city, the home of the 1996 Summer Olympic Games.

The success of a conference requires a lot of commitment and enthusiasm from many people, in particular, my organizing committee and the steering committee. First of all, I express my most sincere appreciation to Yan Solihin, the program chair, who worked extremely hard with his first-class program committee and put together a very strong technical program this year. I would like to thank my long-time friend Murali Annavaram who agreed to serve the role of finance chair given this very difficult time in meeting our budget goal and getting external financial support. I also thank my local arrangement chair Rich Vuduc who helped me working with the hosting hotel Georgia Tech Hotel during the planning stage. I would like to thank Xipeng Shen, the publication chair, for negotiating with the publisher and keeping a close eye on the authors of all accepted papers for the Proceedings preparation. I also extend my appreciation to Tom Wenisch who put together two interesting, timely tutorials on simulation tools and cloud computing workloads. I would like to thank Huiyang Zhou who maintains the electronic registration website and will run the registration desk during the conference period. This job will not end until the last day of the conference. I would like to thank Engin Ipek and Zhibin Yu for making sure people in the community are well informed about IISWC-2010. Special thank goes to Byeong Kil Lee who has done a great job in keeping the IISWC-2010 master website up to date and I thank Dong Hyuk Woo for helping set up, test-drive, and manage the paper submission website, the most critical part of conferences. I would like to express my gratitude to the steering committee of IISWC-2010, in particular, the previous steering committee chair Dave Kaeli and the incumbent chair Lieven Eeckhout for their guidance, commitment, and impartial support. Finally, I thank Karsten Schwan for agreeing to give this year's keynote.

IISWC-2010 will not succeed without external financial sponsorship. I would like to thank the generosity of AMD (Dave Christie) and National Science Foundation. Their contributions allow us to keep the registration fee low for student participants and make the social banquet possible.

IISWC-2010 is held, the very first time, back-to-back at the same location with another major conference in computer architecture, the 43rd International Symposium on Microarchitecture. We look forward to seeing many of you in both conferences and sincerely hope you will enjoy your interactions with attendees in your research areas and have a rewarding experience in IISWC-2010 at Atlanta, Georgia.

Hsien-Hsin S. Lee
Georgia Institute of Technology
IISWC-2010 General Chair

Message from the Program Chair

I am honored to present the technical program of the 2010 IEEE International Symposium on Workload Characterization (IISWC-2010). It has been IISWC tradition to uniquely include papers that deal with workload characterization and modeling that span various computing layers and platforms. IISWC 2010 is not an exception. This year's program includes 21 regular papers, 5 posters, a keynote speech by Karsten Schwan, and a panel.

IISWC 2010 received 56 paper submissions. Each paper received at least three reviews, with most of them receiving four or five reviews. At least three reviews came from the program committee members, with one or two additional reviews coming from external reviewers, solicited by program committee members. The review process was performed in a double blind manner, where the identity of authors were hidden from reviewers, and the identity of reviewers were hidden from authors. After all reviews were collected, authors were allowed to respond to reviews during a rebuttal period.

The PC meeting was held on August 21, 2010 in NCSU campus, Raleigh, North Carolina. 19 of the 22 program committee members (representing 87% of total) attended the meeting, mostly in person, and some through teleconferencing. The meeting started at 8:30am and lasted until 5:30pm. In the meeting, right before when each paper was discussed, program committee members who had a conflict of interest were excused from the room. Then, the authors' names were revealed to the program committee, and the program committee members who reviewed the papers discussed it and arrived at a decision. Only when the PC members could not arrive at a decision, voting was opened to other PC members. For papers that I have conflict of interest with, John Carter and Lieven Eeckhout handled the reviewing process and the PC meeting discussion. In the end, the PC decided to accept 21 regular papers, and 5 posters. This represents an acceptance rate of 37.5% for regular papers.

I would like to express my gratitude to paper authors who submitted their work to IISWC and to all PC members who worked hard and spent many hours reviewing and selecting papers. I also thank the IISWC organizing committee led by Hsien-Hsin Lee, and the steering committee, for their advice and help.

Yan Solihin
NC State University

IISWC 2010 Organizing Committee

Committees

General Chair
Hsien-Hsin S. Lee, *Georgia Tech*

Program Chair
Yan Solihin, *NC State University*

Workshop/Tutorial Chair
Tom Wenisch, *University of Michigan*

Finance Chair
Murali Annavaram, *USC*

Registration Chair
Huiyang Zhou, *NC State University*

Publicity Co-Chairs
Engin Ipek, *University of Rochester*
Zhibin Yu, *Huazhong U. of Sci. & Tech*

Publications Chair
Xipeng Shen, *College of William & Mary*

Web Chair
Byeong Kil Lee, *UTSA*

Submission Chair
Dong Hyuk Woo, *Intel Labs*

Local Arrangements
Richard Vuduc, *Georgia Tech*

Program Committee
Tor Aamodt, *U. of British Columbia*
Rajeev Balasubramoniam, *U. of Utah*
Gordon Bell, *IBM*
John Carter, *IBM*
Sangyeun Cho, *U. of Pittsburgh*
Jaewoong Chung, *AMD*
Nate Clark, *Georgia Tech*
Lieven Eeckhout, *Ghent University*
Weng Fai Wong, *Nat'l U. of Singapore*
Sudhanva Gurumurthi, *U. of Virginia*
Scott Hahn, *Intel*
Seongbeom Kim, *VMware*
Keiji Kimura, *Waseda University*
Victor W. Lee, *Intel*
Tao Li, *U. of Florida*
Jude Rivers, *IBM Research*
Brian Rogers, *IBM*
Li Shang, *U. of Colorado*
Xipeng Shen, *College of William & Mary*
James Tuck, *NCSU*
Richard Vuduc, *Georgia Tech*
Li Zhao, *Intel*

Steering Committee
Pradip Bose, *IBM Research*
Derek Chiou, *U. of Texas at Austin*
David Christie, *AMD*
Tom Conte, *Georgia Tech*
Lieven Eeckhout, *Ghent University*
Jay Jayasimha, *Intel*
Lizy John, *UT-Austin*
David Kaeli, *Northeastern University*
Alan Lee, *AMD*
David Lilja, *University of Minnesota*
Ann Marie Maynard, *IBM*
Onur Mutlu, *CMU*
Ravi Nair, *IBM*
John Shen, *Nokia*
Ben Zorn, *Microsoft*

IISWC 2010 Reviewers

Tor Aamodt
Rajeev Balasubramoniam
Gordon Bell
Paul Brett
John Carter
Vineet Chadha
Niladrish Chatterjee
Sangyeun Cho
Jaewoong Chung
Socrates Demetriades
Lieven Eeckhout
Fei Guo
Ziyu Guo
Sudhanva Gurumurthi
Scott Hahn
Anthony Hylick
Xiaowei Jiang
Seongbeom Kim
Keiji Kimura
Victor W. Lee
Tao Li
Bin Lin
Gabriel Loh
Michael Mesnier
Michael Moeng
Seth Pugsley
Dheeraj Reddy
Jude Rivers
Brian Rogers
Li Shang
Xipeng Shen
Kshitij Sudan
James Tuck
Richard Vuduc
Yasutaka Wada
John-David Wellman
Weng Fai Wong
Li Zhao
Eddy Zhang

Table of Contents

SESSION 4: Parallelism

SESSION 5: Virtualization and Cloud

SESSION 6: Power, Network, and Multi-threading/Multicore

List of Posters

Keynote Talk

Cloud and Utility Computing and its Implications for Large-scale Datacenters

Karsten Schwan
Director of CERCS Research Center, Georgia Tech

Abstract:

Cloud computing and Internet applications are driving forces behind the ever larger datacenters constructed in the U.S. and worldwide. This talk presents some of these trends and identifies some of the challenges for datacenter management they are causing. Challenges include scale, where without automation, it becomes impossible to manage datacenter facilities, assess current state, and understand management options and tradeoffs. Another challenge is increased dynamics, where cloud applications used to exploit spare resources are causing increased dynamics in workload, and where increased internationalization can hide the diunral variations typically seen in national settings. A final challenge presented in this talk is due to the continued evolution in IT platforms, where power/performance improvements are gained by increased use of heterogeneity in processors and memory, making it difficult for applications to exploit system resources and evenly distribute their loads across datacenter machines.

Bio:

Karsten Schwan is a Regents' Professor in the College of Computing at the Georgia Institute of Technology. He also a Director of the Center for Experimental Research in Computer Systems (CERCS), with co-directors from both GT's College of Computing and School of Electrical and Computer Engineering. The NSF-sponsored CERCS research center's faculty conduct research in experimental computer systems in the domains of Enterprise, High Performance, and Embedded/Pervasive Systems, with members from industry and from federal agencies. Prof. Schwan's M.S. and Ph.D. degrees are from Carnegie-Mellon University in Pittsburgh, Pennsylvania, where he began his research in high performance computing, addressing operating and programming systems support for the Cm* multiprocessor. At the Ohio State University, he established the PArallel, Real-time Systems (PARTS) Laboratory, containing both custom embedded processors and commercial parallel machines, and conducting research on operating and programming system support for cluster computing and for adaptive real-time systems. At Georgia Tech, his work ranges from topics in operating and communication systems, to middleware, to parallel and distributed applications, focusing on information-intensive distributed applications in the enterprise domain (e.g., the operational information systems supporting large enterprises) and in the high performance domain (e.g., high performance I/O and remote data visualization). Technical topics currently pursued in his research span (1) scalable techniques for virtualizing and managing future multi-core and

multi-machine platforms, (2) efficient methods for managing applications and services in datacenter and cloud computing systems, including new techniques for runtime performance and behavior monitoring and understanding, (3) middleware for high performance data movement, addressing I/O in future petascale machines and QoS-sensitive data streaming in pervasive and wide area systems, and (4) experimentation with representative applications in the HPC, enterprise, and pervasive domains.

Analysis on Semantic Transactional Memory Footprint for Hardware Transactional Memory

Jaewoong Chung
Intel Labs, Santa Clara, USA
jaewoong.chung@intel.com

Dhruva R. Chakrabarti
HP Labs, Palo Alto, USA
dhruva.chakrabarti@hp.com

Chi Cao Minh
chi.caominh@stanfordalumni.org

***Abstract*—We analyze various characteristics of *semantic* transactional memory footprint (STMF) that consists of only the memory accesses the underlying hardware transactional memory (HTM) system has to manage for the correct execution of transactional programs. Our analysis shows that STMF can be significantly smaller than *declarative* transactional memory footprint (DTMF) that contains all memory accesses within transaction boundaries (i.e., only 8.3% of DTMF in the applications examined). This result encourages processor designers and software toolchain developers to explore new design points for low-cost HTM systems and intelligent software toolchains to find and leverage STMF efficiently. We identify seven code patterns that belong to DTMF, but not to STMF, and show that they take up 91.7% of all memory accesses in transactional boundaries, on average, for the transactional programs examined. A new instruction prefix is proposed to express STMF efficiently, and the existing compiler techniques are examined to check their applicability to deduce STMF from DTMF. Our trace analysis shows that using STMF significantly reduces the ratio of transactions overflowing a 32KB L1 cache, from 12.80% to 2.00%, and substantially lowers the false positive probability of Bloom filters used for transaction signature management, from 23.60% to less than 0.001%. The simulation result shows that the STAMP applications with the STMF expression run 40% faster on average than those with the DTMF expression.**

I. Introduction

Multi-core processors are now common in server, client, and even embedded systems. However, the complexity of parallel programming prevents programmers from taking full advantage of multiple cores due to cumbersome issues such as deadlocks, data races, and synchronization tradeoffs. Transactional Memory (TM) [7], [13], [15], [20] offers a promising solution for parallel programming. With TM, programmers simply mark a section of code as a transaction. TM systems guarantee that the instructions are executed atomically (i.e., all or no instructions are executed) and in isolation (i.e., no intermediate results of the instructions are exposed to the rest of the system). Transactions are allowed to run in parallel unless they access the same address and either of them writes to the address (i.e., transaction conflict).

While the wide acceptance of TM in academia and industry led to the development of first-generation TM hardware support in commercial systems such as SUN's Rock processor [11], Azul system [3], and AMD's Advanced Synchronization Facility [12], it is a significant challenge for processor designers to justify a sizable amount of hardware resources for TM mainly due to the "chicken and egg" problem [9]. There are few transactional programs to justify the transistor budget and verification cost of hardware TM implementations. On the flip side, hardware support for transactional programming platform has barely started to emerge in commercial systems.

In investigating existing transactional programs to see how many transactions an hardware TM (HTM) system can support with limited hardware resources, we made an interesting observation from transactional programs optimized manually for software TM (STM) systems [5], [21]–[23]. Unlike HTM systems that use dedicated hardware for transactional execution, STM systems instrument the code with software barriers to track memory accesses in transactions. To mitigate the runtime overhead of the barriers, advanced programmers eliminate unnecessary barriers from the transactional programs. While only a subset of memory accesses in transactions is managed with barriers in STM systems, program correctness is not compromised since the rest of the transactional memory accesses neither generate transaction conflicts nor produce transactional data to manage speculatively (e.g., accesses to read-only data). To see how many memory accesses in transaction boundaries demand the underlying TM system's management, we analyzed the manually optimized version of STAMP [5], a TM benchmark suite used widely for TM architecture studies. To our surprise, the results showed that about 91.7% of memory accesses in transactions are not instrumented with barriers. It indicates that the *semantic* transactional memory footprint (STMF) that consists of only the memory accesses the underlying TM system has to manage for the correct execution of transactional programs can be much smaller than *declarative* transactional memory footprint (DTMF) that counts all memory accesses in transaction boundaries.

In this paper, we analyze various characteristics of STMF for HTM systems. The main goal of this work is to help software toolchain developers and processor designers understand the architectural implications of STMF and encourage them to explore new design points for low-cost

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

HTM systems and intelligent software toolchains to find and leverage STMF efficiently. We investigate the source code of existing transactional programs and use a memory trace analyzer and a TM simulator to answer the following questions.

- **Why is STMF much smaller than DTMF?** We identified seven types of transactional memory accesses that belong only to DTMF such as immutable data and stack accesses. They take up 91.7% of all transactional memory accesses in the applications we examined.
- **How is a TM ISA extended to enable software to express STMF efficiently?** We present a new approach to express STMF with an instruction prefix. Unlike the previous proposals that add new memory instructions to allow for non-transactional memory accesses in transaction boundaries [15], [16], [27], the instruction prefix can be selectively annotated to any existing instructions to make the implicit/explicit memory operations associated with them transactional, which is very useful for the CISC architecture with complex instructions such as the x86 ISA [1].
- **What are the compiler techniques to identify STMF?** We examined the existing compiler techniques including those developed for STM barrier optimization to see if they can be used to identify STMF.
- **What is the architectural implication of STMF to cache-based HTM designs?** For an HTM with a 32KB L1 cache, transactional programs with the STMF expression decreases transaction overflows from 12.80% to only 2.00%. For a configuration that mimics the SUN Rock processor [11], the ratio of overflowed transactions also declines significantly, from 42.59% to 6.70%.
- **What is the architectural implication of STMF to signature-based HTM designs?** For signature-based schemes, STMF reduces the false positive probability of Bloom filters [4] substantially, from 23.60% to less than 0.001% in comparison to DTMF.
- **What is the overhead of expressing STMF in the binary code?** The code size increment due to the instruction prefix is negligible. It adds only up to 251 bytes to the executable files of the tested applications.
- **What is the performance impact of STMF in comparison to DTMF?** The evaluation result with the STAMP applications shows that the transactional code with the STMF expression improves performance by 40% on average for the same amount of TM hardware resources.

The rest of the paper is organized as follows. Section II explains transactional memory briefly. Section III analyzes the source code of transactional programs and explains why STMF is much smaller than DTMF. Section IV describes a new ISA extension to express STMF and examines the applicability of the existing compiler techniques for STMF. Section V and Section VI present the experimental results with a trace analyzer and a TM simulator respectively. Section VII discusses related work, and Section VIII concludes the paper.

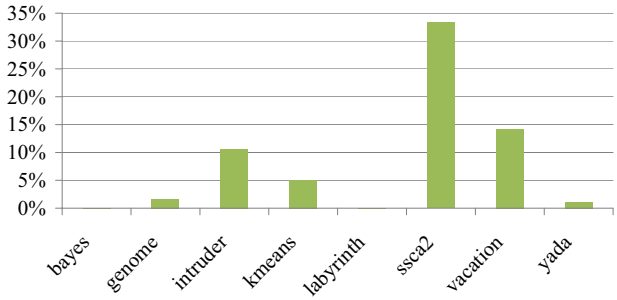


Figure 1. The ratios of the memory accesses instrumented with software barriers to all memory accesses in transactions.

II. Hardware Transactional Memory

Transactional Memory (TM) makes it easy for programmers to develop parallel programs. With TM, programmers enclose a group of instructions within a transaction to execute them in an atomic and isolated way. The underlying TM system runs transactions in parallel as long as they do not conflict. Two transactions conflict when they access the same address and one of them writes to it. An HTM system uses dedicated hardware to accelerate transactional execution [7], [13], [15], [20]. It starts a transaction by taking a register checkpoint with shadow register files. Whenever the transaction writes to memory, the transactional data produced by the write operation are maintained separately from the old data by either buffering the transactional data in hardware buffers, such as the cache, or logging the old value (i.e., data versioning). It augments the cache with additional bits or uses separate hardware structures such as Bloom filters to record the memory addresses read by the transaction in its read-set and those written in its write-set. A conflict between two transactions is detected by comparing the read-sets and the write-sets of both transactions (i.e., conflict detection). If a conflict is detected, the transaction is rolled back by undoing the transactional write operations, restoring the register checkpoint, and discarding the transactional metadata (i.e., the read-/write-sets). Absent a conflict, the transaction ends by committing the transactional data and discarding the transactional metadata and the register checkpoint.

III. Source Code Analysis

A. Semantic Transactional Memory Footprint

While studying existing transactional programs, we noticed that the transactional programs optimized manually for STM systems use software barriers only for a small

Application	Input Parameter
bayes	-v32 -r4096 -n2 -p20 -s0 -i2 -e2 -t8
genome	-g512 -s32 -n32768 -t8
intruder	-a10 -l16 -n4096 -s1 -t8
kmeans	-m40 -n40 -t0.05 -i random-n16384-d24-c16.txt -p8
labyrinth	-i random-x48-y48-z3-n64.txt -t8
ssca2	-s14 -i1.0 -u1.0 -l9 -p9 -t8
vacation	-n2 -q90 -u98 -r1048576 -t4096 -c8
yada	-a10 -i ttimeu10000.2 -t8

Table I. Input parameters used for STAMP applications.

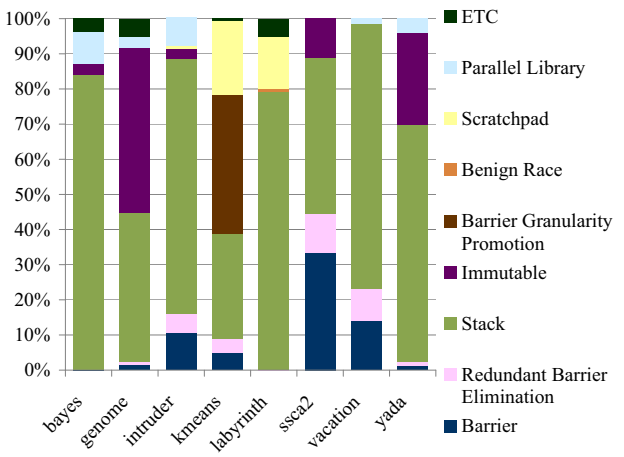


Figure 2. The breakdown of memory access types in transactions.

subset of memory accesses in transactions without hurting program correctness. To measure the ratio of the memory accesses instrumented with software barriers to all memory accesses in transactions, we used the STAMP benchmark suite [5]. It includes eight TM applications presenting several application domains and a wide range of transactional execution cases. **Bayes** implements a hill-climbing algorithm for learning the structure of a Bayesian network. **Genome** performs the genome assembly process. **Intruder** scans network packets for matches against a known set of intrusion signatures. **Kmeans** executes the K-means algorithm that groups objects in an N-dimensional space into K clusters. **Labyrinth** finds the path to the exit in a maze with a hill-climbing algorithm. **Ssca2** has four kernels that operate on a large, directed, weighted multi-graph. **Vacation** implements an on-line transaction processing system. **Yada** implements the Delaunay mesh refinement [5].

We compiled the applications with GCC 3.4.0 and ran them with the large input sets described in Table I on eight x86 cores. The beginning and the end of a transaction are mapped to the acquisition and the release of a single global lock. This mapping produces a serializable transaction schedule and guarantees that we do not measure numbers from aborted and retried transactions, which are influenced by a specific TM implementation scheme. PIN [17] is used to count the number of bytes accessed with and without software barriers in a transaction. Figure 1 shows that the ratio goes down to 0.03% for bayes and 0.01% for labyrinth. The highest ratio was 33% for ssca2. On average, the ratio is only 8.3%.

The low percentage of memory accesses with barriers clearly indicates that the STMF of transactional programs can be much smaller than their DTMF.

B. Why is STMF much smaller than DTMF?

To understand why many memory accesses belong to DTMF, but not to STMF, we ran the STAMP applications on eight x86 cores, as explained in Section III-A, and analyzed different types of memory accesses in transactions. Table I shows the input parameters for the STAMP applications in our experiment. We manually added special markers to the applications and developed several PIN tools that use the markers and other runtime information to categorize memory accesses in transactions into nine types. Figure 2 shows the breakdown of the nine types. The access types other than **barrier** correspond to the memory accesses that do not have software barriers associated with them. We explain why program correctness is preserved even though they are processed non-transactionally.

Barrier represents the memory accesses instrumented with software barriers. There are two kinds of software barriers. A *global* barrier is used for shared data and performs both data versioning and conflict detection. A *local* barrier is used for thread-local data and does not perform conflict detection since other threads do not access the data. Local barriers incur less runtime overhead for STM systems than global barriers because they do not execute conflict detection code. We count both barriers as the same type since few local barriers are found in the STAMP applications, which makes it uninteresting to discern local barriers to consider special hardware support for them. On average, this type represents 8.3% of memory accesses in transactions.

Redundant barrier elimination represents the memory read accesses whose addresses have already been accessed by other memory read accesses with barriers. Since the addresses are already protected with barriers, optimization opportunities exist for the subsequent references. The STAMP applications do not remove barriers for the rest of redundant barrier patterns, such as a read barrier reading the same address protected already by a write barrier in order to work with an STM system that buffers transactional data at every write (i.e., lazy data versioning). Redundant barrier elimination helps reduce barrier runtime overhead for STM systems. However, it does not help reduce TM hardware resource consumption for HTM systems since hardware resources are consumed per memory *addresses* accessed by transactions, not by the number of memory *accesses* to the

addresses in HTM systems. This type constitutes 3.9% of memory accesses in transactions on average.

Stack represents the memory accesses to two kinds of stack variables that do not escape the stack. The first kind is the stack variables allocated *after* a transaction begins. The second kind is the stack variables allocated *before* a transaction begins but initialized by the transaction before they are used. The former do not require data versioning since variable allocation is undone by restoring the stack pointer when the register checkpoint is restored at a transaction abort. The latter do not require data versioning since they are overwritten by the restarted transaction. The stack variables that stay in the stack do not need conflict detection since the stack is a thread-local memory space. In the STAMP applications, all stack variables belong to one of the two types and do not require barriers. This is the dominant type of barrier-free memory accesses, accounting for 61.9% of memory accesses in transactions on average.

Immutable represents the memory accesses to the data that are read-only after threads are spawned. 47.1% of memory accesses in genome are of this type. Genome processes a lot of DNA sequences, and they are all read-only. Read-only data are easily found as input datasets in scientific applications as well [24]. This type constitutes 11.3% of memory accesses in transactions on average.

Barrier granularity promotion represents the memory accesses to a memory location that should be accessed with barriers semantically, but are accessed without barriers since an alternative memory location is protected instead with barriers for conflict detection. Application code ensures that a conflict is detected at the alternative memory location if one is to be detected at the original memory location. An example is detecting conflicts at object headers instead of object fields. An example found in our analysis is from kmeans, in which 39.5% of memory accesses are of this type. Kmeans passes around the array structures to calculate the K means of input data. Since all elements of the arrays are written or read together, a single barrier is used to protect only the first element of the array when the array is read. A conflict against another transaction that writes to the array is detected when the transaction writes to the first element. On average, 4.9% of memory accesses are of this type.

Benign race leverages the programmer's understanding on application semantics. This type of memory accesses is found only in labyrinth, in which an optimistic concurrency control mechanism is implemented in the application code. A worker thread reads shared information about interim solutions to exit the maze. The read operation is executed without barriers. Once the worker thread calculates its own solution based on a hill-climbing algorithm [5], it compares the current shared information with what it previously read. A difference indicates that another worker has updated the shared information. In this case, the worker thread drops its solution and restarts from the beginning. This approach is useful when the implemented algorithm does not care about races as long as the memory values of interest stay as they were. In other words, it ignores the case in which the value of a variable changes from A to B and then back to A (i.e., the ABA problem). 0.8% of memory accesses in labyrinth are of this type.

Scratchpad represents memory accesses to the thread-local heap locations that transactions always write first, then read. These memory locations are used as scratchpads to calculate and temporarily contain intermediate results. For the cases found in our analysis, stack variables are not useful for this purpose because the intermediate results are accessed by multiple functions that do not have a caller-callee relationship. Data versioning is not required for scratchpad variables since they are initialized before being used. 21.1% of memory accesses in kmeans are of this type. Kmeans generates these memory accesses while accessing some temporary locations for calculating means. On average, 4.6% of memory accesses are of this type.

Parallel library represents the memory accesses generated by parallelized libraries. The STAMP applications use a concurrent memory manager that supports multithread-safe malloc() and free() with per-thread memory pools. If an object is allocated in a transaction and the transaction is rolled back, the object is discarded as garbage. This is a memory leak but did not cause a memory problem in our experiments. 13.1% of memory accesses in bayes are of this type. It has many malloc() calls to allocate vector structures for building a Bayesian network [5]. The vector structures are used to contain the edges weighted with probabilities for the network. 4.3% of memory accesses are of this type on average.

ETC represents the memory accesses that do not belong to any of the types above. On average, only 1.8% of memory accesses are of this type.

Overall, there are various types of memory accesses that require neither data versioning nor conflict detection for correct transactional execution. They constitute a significant portion of memory accesses in transactions (e.g., about 91.7% for the STAMP applications), and do not compromise the correctness of transactional programs. They belong only to DTMF and make STMF much smaller than DTMF.

IV. ISA Extension, Implementation, and Compiler Support

In this section, we present an instruction prefix to express STMF efficiently for HTM systems. Then, we explain an implementation scheme for the ISA and compiler techniques for STMF.

A. TM ISA Extention for STMF

Table II shows a simple HTM ISA extension to express STMF by selectively annotating transactional memory accesses with an instruction prefix. It supports two instructions,

Category	Mnemonic	Description
Transaction Demarcation	TxBegin [pc]	Start a transaction. Jump to [pc] when the transaction is rolled back.
	TxEnd	End the current transaction.
STMF Expression	TX Prefix	Make the prefixed instruction transactional.

Table II. Hardware TM ISA for selective annotation.

```
UpdateMeans:
  TxBegin [UpdateMeans]
  MOV rAX, K
Loop:
  MOV rBX, [Scratchpad + rAX* scale]
  TX MOV [SharedArray + rAX * scale], rBX
  DEC rAX
  JNZ [Loop]
  TxEnd
  RET
```

Figure 3. Code example with selective annotation.

TxBegin and TxEnd, to demarcate transaction boundaries like existing HTM ISAs typically do [11], [18]. TxBegin starts a transaction and takes an alternative address as argument. If the transaction is rolled back, the execution flow jumps to the alternative address once the rollback procedure completes.

To express STMF efficiently, we introduce an instruction prefix, TX. An instruction prefix makes it easy to add a new attribute to existing instructions. For example, the LOCK prefix in the x86 architecture [1] makes the memory operations of prefixed instructions atomic. We propose that TX be prefixed to instructions in transaction boundaries to make the implicit/explicit memory operations associated with the instructions transactional. The underlying HTM system processes those memory operations transactionally through data versioning (e.g., buffering transactional data or logging old data) and conflict detection (e.g., using cache coherence protocol). The instructions without the TX prefix are executed without consuming any TM hardware resources and committed as soon as they retire regardless of transaction boundaries.

A transaction is committed with TxEnd or rolled back due to conflicts. TxEnd ends the transaction by committing transactional data and discarding transactional metadata. If the transaction is rolled back, the memory operations done by the instructions with the TX prefix are undone, and the register checkpoint is restored. However, the memory operations done by the instructions without the TX prefix are left done. It is the responsibility of compilers, managed runtimes, or advanced programmers to guarantee the semantic correctness of the transactions with the un-prefixed instructions.

Figure 3 shows a pseudo-code example using the proposed ISA based on the x86 architecture. The code mimics part of the kernel code in kmeans, in which it copies the K newly calculated means from a thread-local scratchpad (i.e., *Scratchpad*) to a shared array (i.e., *SharedArray*). *Scale* is the size of each array element that contains information about a newly calculated mean. The key point of the example is that the MOV operations to load data from the scratchpad do not need the TX prefix since they read from a thread-local memory space. This reduces TM hardware resource consumption by half relative to the case in which both MOV operations to load from the scratchpad and store to the shared array are processed transactionally.

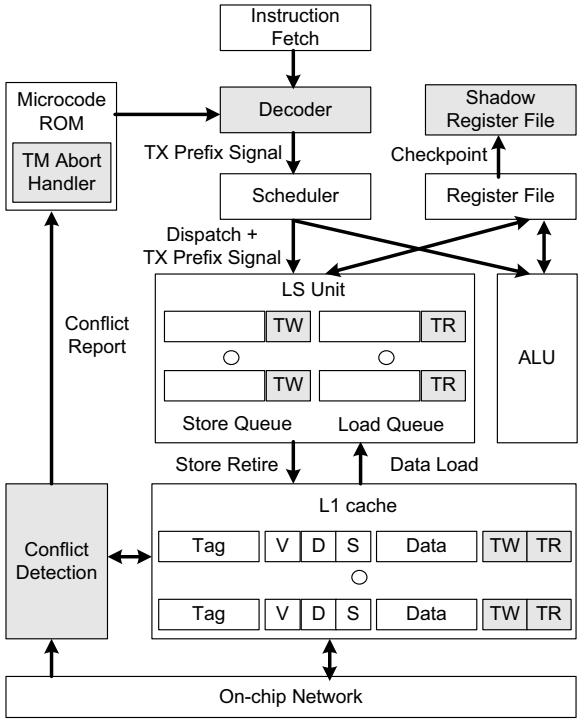


Figure 4. An ISA implementation for STMF. The components changed or added for the ISA implementation are shown in gray.

The ISA allows the same cache line to be accessed by both transactional accesses and non-transactional accesses. A transactional access following a non-transactional access to the same address is allowed since the non-transactional access is committed when the instruction triggering the access retires. A non-transactional access following a transactional access to the same address is silently made to be transactional at the cost of increasing STMF.

B. ISA Implementation

Figure 4 shows a cache-based implementation scheme of the proposed ISA. In the figure, the components changed or added for the implementation are shown in gray. The implementation uses the cache as buffer for transactional data (lazy data versioning) and uses the cache coherence

protocol for conflict detection (eager conflict detection). It adds two bits per cache line: the TW (transactional write) bit for transactional stores and the TR (transactional read) bit for transactional loads. The TW bit is also added per store queue entry, and the TR bit per load queue entry. The implementation has a shadow register file for register checkpoint. The decoder logic has been changed to recognize TxBegin, TxEnd, and the TX prefix.

TxBegin starts a transaction by taking a register checkpoint and storing the alternative PC as part of the register checkpoint to jump to it when the transaction rolls back. Detecting the TX prefix, the instruction decoder sends a signal to the Load/Store (LS) unit when the prefixed memory operation is dispatched. Receiving the signal, the LS unit sets the TW bit for a store operation and the TR bit for a load operation in the corresponding queue entry. The TR bit is cleared when the prefixed instruction retires, and the corresponding TR bit in the cache is set by then. The TW bit is cleared when the store is sent to the cache, and the TW bit in the cache is set. The transactional data are buffered in the cache. A transaction conflict is detected by comparing the cache coherence messages against the TW/TR bits in the cache and the portion of the store queue that contains the store operations of retired instructions. A coherence message conflicts against the bits in two cases. The first occurs when the message is for data invalidation and the TW bit or TR bit is set for the matching address. The second occurs when the message is for data sharing and the TW bit is set for the matching address. The microcoded transaction abort handler is invoked when a conflict is detected. The handler invalidates the cache entries with the TW bits, clears all TW/TR bits, restores the register checkpoint, and flushes the pipeline. The execution flow starts from the alternative PC at the next cycle. If the transaction reaches TxEnd, it is committed by clearing all TW/TR bits and discarding the register checkpoint.

C. Compiler Support for STMF

STMF can be deduced from DTMF at compile time. Table III shows the seven access patterns that do not belong to STMF, the applicability of compiler techniques to identify the patterns, and the usefulness of the techniques for HTM systems.

Stack: It is well understood in the compiler literature how to identify stack variables and check if the variables escape the stack [23]. Once the variables are confirmed to stay in the stack, the compiler looks for two categories of stack variables to remove from STMF. The first category consists of stack variables allocated after a transaction begins and are easily identified by examining their scope. The second category corresponds to variables allocated outside a transaction but initialized before used within a transaction. This scenario can be detected by the compiler by verifying that every read from a stack variable within a transaction is dominated by a write to the same stack variable within the same transaction. As explained in Section III-B, these variables are not included in STMF. There is a third category of stack variables that may not be included in STMF: those allocated and initialized outside a transaction but used in a transaction. If there is no write to such a stack variable within the transaction, the variable can be removed from STMF. Otherwise, data versioning is still required to be able to revert back the correct value of the stack variable at a transaction abort, and hence the stack variable is added to STMF. Our experience with STAMP applications shows that this last category is very rarely encountered.

Access Patterns	Compile Time Identification	Usefulness to identify STMF
Stack	Yes	Yes
Immutable	Yes	Yes
Barrier Granularity Promotion	Yes	Yes
Scratchpad	Yes	Yes
Benign Race	No	Yes
Parallel Library	No	Yes
Redundant Barrier	Yes	No

Table III. Applicability of compiler techniques to identify STMF for HTM systems.

Immutable: Immutable data can be found by searching for read-only data or write-once data [2]. The compiler can leverage the language specifications guaranteeing read-only or write-once properties (e.g., const in C/C++) to find and exclude them from STMF.

Barrier granularity promotion: In object-oriented languages such as Java, the member fields of an object are accessed by traversing through the header of the object [14]. The compiler can identify the memory operations to access the member fields in a transaction and generate the code that accesses the header transactionally and the member fields non-transactionally. This puts only the header in STMF but leaves the member fields in DTMF. As a result, the underlying TM system gets to manage transactional data at object granularity with a smaller STMF at the higher risk of increasing false transaction conflicts. This optimization works only with eager conflict detection, in which a transaction is aborted as soon as a transaction conflict is detected with the access to the header [20]. The early conflict detection prevents the aborted transaction from accessing the member fields non-transactionally. In addition, the compiler needs to make sure that the generated code accesses the header before accessing the member fields in the same transaction.

Scratchpad: As explained in Section III-B, neither data versioning nor conflict detection is required for the variables used as scratchpads. The compiler analysis necessary to identify them is very similar to that used for stack variables,

but performed inter-procedurally to make sure that they stay in thread-local heaps.

Benign race and **parallel library**: These two types are hardly handled by the compiler. We envision that parallel libraries for transactional programs are optimized manually by advanced programmers and commonly used by average programmers [6].

Redundant barrier: There are many STM compiler techniques to optimize redundant barriers to the same address, thus reducing the runtime overhead of executing software barriers [2], [23]. However, these techniques are not useful for reducing the size of STMF since STMF is decided by the number of unique memory addresses accessed, not how many times each address is accessed.

Note that the compiler techniques we described above are most effective when performed across procedures (i.e., inter-procedural analysis) and the whole program is available at compile time (i.e., whole-program mode). However, this scenario is often not realistic. In addition, most compilers support only intra-procedural optimizations by default. As a consequence, the opportunities to identify STMF at compile time might not be fully leveraged. Fortunately, the stack accesses that account for the major part of the difference between DTMF and STMF (i.e., 61.9% in our study) can be handled easily by intra-procedural analysis [23].

V. Trace Analysis

To understand the architectural impacts of STMF, we developed multiple PIN tools and ran the STAMP applications [5] with them. For each application, the STAMP benchmark suite provides two versions that have the same transactions: one for HTM and the other for STM. The HTM version has only the markers to express the beginning and the end of a transaction. The STM version has additional software barriers to selectively annotate transactional accesses in the transaction. We used the HTM version for the transactional program with the DTMF expression (i.e., only transaction demarcation with TxBegin and TxEnd), and the STM version for the transactional program with the STMF expression after substituting the software barriers with the TX prefixes.

The applications ran on eight x86 cores in the same way explained in Section III. The beginning and the end of a transaction are mapped to the acquisition and the release of a single global lock. This scheme produces a serializable transaction schedule and simplifies the development of PIN tools for multiple threads because the statistics from transactions are collected sequentially.

A. STMF Influence to Cache-based HTMs

To examine how many transactions fit into a specific cache organization with DTMF and STMF, we ran the applications with three transactional buffer configurations. The first configuration uses a 32KB 4-way L1 cache with 32-byte cache line as a transactional buffer, which is the same as the implementation presented in Section IV-B. The second configuration doubles the buffer size by using a 64KB 4-way L1 cache with 32-byte cache line. The third configuration mimics the buffer organization of the SUN Rock processor, in which a 32-entry store queue is used for transactional store operations and a 32KB 4-way L1 cache for transactional load operations [11]. Table IV shows the ratios of transactions that fit into the buffers used in each configuration with DTMF and STMF. 100% means that all transactions are supported by the given buffer size without transactional data overflow.

In the table, 87.20% of transactions fit into the 32KB L1 cache with the DTMF expression on average. This confirms the observations from a previous study that the majority of transactions are likely to be short-lived even for realistic transactional programs [9]. However, 12.80% of the transactions still overflow the cache. These are long-lived transactions whose memory footprints are exponentially larger than short-lived ones [8], [10]. Working with a slow TM overflow scheme assuming that transaction overflows are rare, these long transactions may cause significant performance degradation. One solution to avoid the performance degradation is to increase the cache size. Table IV shows that the 64KB L1 cache decreases the ratio of overflowed transactions only by 3.08%, insignificant given that hardware costs almost doubled. On the other hand, the STMF expression increases the ratio of transactions fitting into the cache significantly – 98.00% on average – with the same buffer size (32KB L1 cache). It also makes 100% of transactions fit into the cache for five applications. The performance impact of STMF is even more evident with the configuration that mimics the SUN Rock processor. It uses a 32-entry store queue to contain the transactional data, which is small for most of the applications, as shown in the table. As a result, only 57.41% of transactions fit into the store queue on average. However, the STMF expression doubles the ratio up to 93.30% and makes 100% of transactions fit into the given buffer size for four applications.

B. STMF Influence to Signature-based HTMs

There are HTM proposals that use the signatures of transaction read-/write-sets to detect transaction conflicts (e.g., Bulk [7] and LogTM-SE [25]). They use hardware structures, such as Bloom filters [4], to manage the read/write signatures. Although they handle long-lived transactions without requiring additional hardware resources, signature-based schemes suffer from false conflicts due to address aliasing in the hardware resources. The STMF expression helps reduce the false positive probability by reducing the size of the read-/write-sets to be recorded in the hardware structures. We first counted the number of unique memory addresses accessed transactionally in transactions. Then, assuming a system that uses two 512-bit Bloom filters (i.e.,

		bayes	genome	intruder	kmenas	labyrinth	ssca2	vacation	yada	average
Number of Dynamic Transactions		522	19488	54926	87412	144	93709	4096	13924	
32KB L1 Cache	DTMF	82.08%	98.76%	99.65%	99.99%	55.54%	99.97%	83.10%	78.46%	87.20%
	STMF	100%	100%	99.75%	100%	100%	100%	89.87%	94.34%	98.00%
64KB L1 Cache	DTMF	87.67%	99.98%	99.76%	99.99%	55.54%	99.99%	93.11%	86.22%	90.28%
	STMF	100%	100%	99.76%	100%	100%	100%	93.78%	94.90%	98.56%
32-Entry SQ + 32KB L1 Cache	DTMF	39.87%	19.54%	66.51%	100%	54.85%	100%	0.74%	77.74%	57.41%
	STMF	100%	100%	99.76%	100%	78.45%	100%	89.87%	78.32%	93.30%

Table IV. The ratios of transactions that fit into the buffers in three configurations: 32KB L1 cache, 64KB L1 cache, and 32-entry store queue + 32KB L1 cache.

		bayes	genome	intruder	kmenas	labyrinth	ssca2	vacation	yada	average
DTMF	Read Filter	25.72%	7.85%	0.33%	0.37%	94.93%	0.00%	28.09%	31.52%	23.60%
	Write Filter	5.65%	0.13%	0.01%	0.00%	92.78%	0.00%	2.90%	18.40%	14.98%
STMF	Read Filter	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	10.30%	0.00%	1.29%
	Write Filter	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Table V. The false positive probability of Bloom filters for managing transaction signatures with DTMF and STMF.

one for the read set and the other for the write set) and hash functions with uniform probability, we calculated the false positive probability of the filters with the number of hash functions that minimize the probability [19]. Table V shows the false positive probability of the filters with DTMF and STMF. It is a false positive when an address is not inserted to a filter, but the membership check with the filter indicates that the address is present. With the DTMF expression, the average probability of false positives is 23.60% for the read-set signature and 14.98% for the write-set signature. The STMF expression substantially reduces the false positive probability to less than 0.001% for all cases except the read-set for vacation.

C. Prefix Overhead for STMF

The TX prefix incurs two kinds of overheads: static code size increment and dynamic instruction fetch bandwidth increment. First, the cost size of transactional programs is increased statically when the TX prefix is added. We measured the size increment of the executable files that resulted from the addition of 1-byte TX prefixes and found that the increment is negligible for all eight applications. The TX prefix increases the executable file size by a maximum of only 251 bytes (yada), as shown in Table VI. This is because the TX prefix is used only in transactions, and there is a large amount of non-transactional code (i.e., code segments outside transactions) and library code. Even when measuring the size increment of the transactional code only, the TX prefix overhead is quite low. The table shows the average numbers of instruction bytes and prefix bytes in a transaction. It also shows the ratios of the number of the prefix bytes to the number of instruction bytes in transactions. On average, the size of transactional code increases by 3.24%. Kmeans and ssca2 show an increase of about 8%, but their actual byte increments are less than 10 bytes.

Feature	Description
CPU	3GHz, 8 x86 cores
L1 Cache	32 KB, 4-way, 32B line, MESI, write-back, 3 cycle hit time, private
L2 Cache	8 MB, 8-way, 32B block, MESI, write back, 15 cycle latency, shared, 8 banks, bit vector of sharers
Memory	4 GB, 100 cycle latency
Interconnect	Tiled network, 32B links, 3 cycles per hop

Table VII. Parameters for the simulated CMP system.

Second, the TX prefix dynamically increases the bandwidth requirement for fetching instructions while the annotated instructions are executed. Table VI shows the ratios of the total number of instruction bytes and the total number of additional prefix bytes fetched for transactional code. On average, the ratio is only 1.85%, which is even lower than the static code size increment. Ssca2 shows the highest ratios for both code size increment and instruction fetch bandwidth increment (i.e., 6.86%) since it has small kernel codes packed densely with transactional memory accesses.

VI. Simulation Study

We implemented the proposed ISA on a CMP simulator as described in Section IV-B and evaluated it with the parameters in Table VII. Transactional data are buffered until committed. Transaction conflicts are detected with the TW/TR bits in the cache. We implemented a simple contention management policy in which the conflicted transaction (i.e., the one receiving a conflicting cache coherence

		bayes	genome	intruder	kmenas	labyrinth	ssca2	vacation	yada	average
Static Code Size Increment	Instruction Bytes	10949	3527	6059	133	7271	80	6234	13077	
	Prefix Bytes	52	22	223	10	17	7	169	251	
	Ratio	0.47%	0.62%	3.68%	7.52%	0.23%	8.75%	2.71%	1.92%	3.24%
Dynamic Instruction Fetch Byte Increment	Instruction Bytes	81.5MB	76.2MB	49.2MB	29.1MB	251.4MB	2.8MB	30.7MB	431.9MB	
	Prefix Bytes	6.1KB	191.4KB	1.2MB	368.9KB	4.6KB	274.5KB	1.1MB	1.1MB	
	Ratio	0.01%	0.25%	2.42%	1.24%	0.00%	6.86%	3.68%	0.25%	1.85%

Table VI. The table shows the static code size increment and the dynamic instruction fetch bandwidth increment due to the TX prefix. For code size increment, it shows the total original instruction bytes in transactions and the total additional prefix bytes in transactional code. For instruction fetch byte increment, it shows the total number of instruction bytes and the total number of prefix bytes fetched for transactional code.

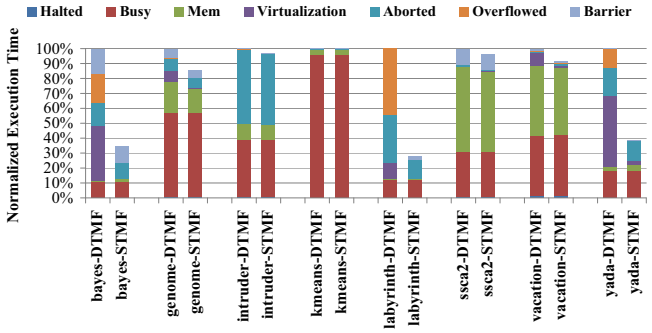


Figure 5. Execution time breakdown of the DTMF version (-DTMF) and the STMF version (-STMF) of the STAMP applications. The bars are normalized to the total execution time of the DTMF version.

message) is aborted. The aborted transaction is restarted after a random back-off delay. A transaction is aborted at buffer overflows as well. We implemented a simple hybrid virtualization mechanism that uses a process-wide commit lock. The commit lock is a software read-write lock accessed non-transactionally. Non-overflowed transactions acquire the lock in read mode immediately after TxBegin. Sharing the read lock, they run in parallel. A transaction overflowing the cache acquires the lock in write mode after restart. The write lock is acquired when there are no read-lock owners. Holding the write lock, the transaction is fine to overflow the cache since the other transactions are waiting to get the lock in read mode after TxBegin. The underlying HTM is notified to ignore cache overflows for the transaction. Though only the overflowing transaction runs while the lock is held in write mode, this simple mechanism properly handles buffer overflows at no additional hardware cost.

Figure 5 shows the execution time breakdown of the DTMF version (-DTMF) and the STMF version (-STMF). The bars are normalized to the total execution time of the DTMF version. In each bar, *halted* is the idle time due to single-threaded code for initialization. *Busy* is for active execution time, *mem* for the stalled time due to memory accesses, *virtualization* for the time to acquire the read-write commit lock, *aborted* for the time wasted due to transaction abort, *overflow* for the execution time of the transactions restarted due to buffer overflow, and *barrier* for the time to synchronize at application barriers.

According to their STMF characteristics, the applications can be categorized into three groups. The first group includes bayes, labyrinth, and yada. This group shows significant performance improvement of about 3x on average with the STMF expression. The applications in this group have large transactions and high ratios of buffer overflows, as shown in Table IV. The speedup comes mainly from eliminating most of the virtualization time and the overflow time by removing buffer overflows. The second group includes genome and vacation. This group has medium-sized transactions and shows performance improvement of 16%. While the STMF expression eliminates the virtualization time and the overflow time effectively, the speedup of the second group is less than that of the first group since the second group has a smaller number of buffer overflows from the beginning. The group average of buffer overflow ratios is 9.07% in Table IV. However, the transaction sizes of these applications grow with larger input data [5], which will make them behave more like the first group. The third group includes intruder, kmeans, and ssca2. This group has very small transactions that fit well into 32K L1 caches, as shown in Table IV. So they make a small performance improvement from the STMF expression when running with 32K L1 caches, 1.8% on average. However, for an HTM design with smaller buffers such as the SUN Rock processor, even these applications may overflow often (e.g., intruder with 33.33% overflowing transactions, as shown in Table IV). The STMF expression can help them run well with such small buffers by reducing the number of buffer overflows.

Overall, transactional programs with the STMF expression ran 40% faster on average than those with the DTMF expression for the applications examined.

VII. Related Work

Researchers have created and analyzed a few TM workloads categorized into two groups: micro-benchmarks and realistic workloads. TM micro-benchmarks typically execute a few operations on data structures such as linked-lists and B-trees [10], [16]. Having small DTMFs, they were

analyzed to run well with a small amount of TM hardware resources and are not likely to take much advantage of the STMF expression, as shown in Section VI. On the other hand, realistic workloads such as BerkeleyDB [10] and the STAMP applications [5] were reported to have large DTMFs and demand a large amount of TM hardware resources for full hardware acceleration of transactional execution. These applications have great potential to require much less hardware when their STMFs are expressed with the appropriate ISA support, such as the TX prefix.

There are proposals for new memory instructions to allow for non-transactional memory accesses in transaction boundaries [15], [16], [27]. All of them can be used to express STMF in the binary code. However, our prefix-based approach for annotating transactional accesses makes it easier to deal with the instructions with memory operands (i.e., those accessing values in memory, not registers), common in modern CISC processors (e.g., x86). The prior art supports specific memory instructions for transactional accesses [15] or non-transactional accesses [16], either of which demands an instruction with memory operands to be split into two or more instructions for the STMF expression (i.e., an equivalent instruction with only register operands and additional special instructions to selectively use transactional/non-transactional accesses for STMF). This can 1) increase the register pressure due to additional use of architectural registers, 2) complicate the compiler implementation for STMF, and 3) can cause atomicity issues by breaking the execution atomicity of the original instruction (e.g., due to an exception/interrupt triggered in between the split instructions). On the other hand, our approach allows instructions with memory operands to access memory transactionally or non-transactionally by simply prefixing them with TX. In comparison to adding new instructions to demarcate non-speculative regions [27], our proposal allows individual non-speculative accesses to be marked easily without requiring non-speculative accesses to be placed in a region. Notary provides an API to privatize and publicize objects for efficient signature-based conflict detection [26]. This API can be used to express STMF in the source code.

VIII. Conclusions

In this paper, we analyzed various characteristics of STMF in transactional programs. We hope that our results help processor designers and software toolchain developers understand the architectural implications of STMF and encourage them to explore new design points for low-cost HTM systems and intelligent software toolchains to find and leverage STMF efficiently.

References

[1] AMD64 Architecture Programmer's Manual. http://developer.amd.com/documentation/guides/Pages/default.aspx.

[2] A.-R. Adl-Tabatabai, B. Lewis, et al. Compiler and Runtime Support for Efficient Software Transactional Memory. In *the Proc. of the 2006 Conf. on Programming Language Design and Implementation*, June 2006.

[3] http://www.azulsystems.com/.

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7), July 1970.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, 2008.

[6] B. D. Carlstrom, A. McDonald, C. Kozyrakis, and K. Olukotun. Transactional Collection Classes, 2007.

[7] L. Ceze, J. Tuck, et al. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *the Proc. of the 33rd Intl. Symp. on Computer Architecture (ISCA)*, June 2006.

[8] J. Chung, C. Cao Minh, et al. Tradeoffs in Transactional Memory Virtualization. In *the Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. Oct. 2006.

[9] J. Chung, H. Chafi, et al. The Common Case Transactional Behavior of Multithreaded Programs. In *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

[10] P. Damron, A. Fedorova, et al. Hybrid transactional memory. In *the Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 2006.

[11] D. Dice, Y. Lev, et al. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS'09: 14th Intl. Conf. on Architectural Support for Programming Languages and Operating System*, 2009.

[12] S. Diestelhorst and M. Hohmuth. Hardware acceleration for lock-free data structures and software-transactional memory. In *www.amd64.org/fileadmin/user_upload/pub/epham08-asf-eval.pdf*.

[13] L. Hammond, V. Wong, et al. Transactional Memory Coherence and Consistency. In *the Proc. of the 31st Intl. Symp. on Computer Architecture (ISCA)*, Munich, Germany, June 2004.

[14] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. of the twenty-second Symp. on Principles of distributed computing*, pages 92–101, New York, NY, USA, July 2003. ACM Press.

[15] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *the Proc. of the 20th Intl. Symp. on Computer Architecture*, May 1993.

[16] S. Kumar, M. Chu, et al. Hybrid Transactional Memory. In *the Proc. of the 11th Symp. on Principles and Practice of Parallel Programming (PPoPP)*, New York, NY, Mar. 2006.

[17] C. Luk, R. Cohn, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *the Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[18] A. McDonald, J. Chung, et al. Architectural Semantics for Practical Transactional Memory. In *the Proc. of the 33rd Intl. Symp. on Computer Architecture*, June 2006.

[19] A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, 2002.

[20] K. E. Moore, J. Bobba, et al. LogTM: Log-Based Transactional Memory. In *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

[21] Y. Ni, A. Welc, et al. Design and implementation of trans-

actional constructs for c/c++. In *OOPSLA '08: Proc. of the 23rd ACM SIGPLAN Conf. on Object-oriented Programming Systems Languages and Applications*, 2008.

[22] T. Shpeisman, V. Menon, et al. Enforcing Isolation and Ordering in STM. In *the Proc. of the Conf. on Programming Language Design and Implementation*, Mar. 2007.

[23] C. Wang, W.-Y. Chen, et al. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *the Proc. of the Intl. Symp. on Code Generation and Optimization*, Mar. 2007.

[24] S. C. Woo, M. Ohara, et al. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, Santa Margherita, Italy, June 1995.

[25] L. Yen, J. Bobba, et al. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *the Proc. of the 13th Intl. Symp. on High Performance Computer Architecture*, Feb. 2007.

[26] L. Yen, S. C. Draper, and M. D. Hill. Notary: Hardware techniques to enhance signatures. In *MICRO '08: Proc. of the 41st IEEE/ACM Intl. Symp. on Microarchitecture*, 2008.

[27] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

Real Java Applications in Software Transactional Memory

Takuya Nakaike, Rei Odaira, Toshio Nakatani, and Maged M. Michael

***Abstract*— *Transactional Memory (TM)* shows promise as a new concurrency control mechanism to replace lock-based synchronization. However, there have been few studies of TM systems with real applications, and the real-world benefits and barriers of TM remain unknown. In this paper, we present a detailed analysis of the behavior of real applications on a software transactional memory system. Based on this analysis, we aim to clarify what programming work is required to achieve reasonable performance in TM-based applications. We selected three existing Java applications: (1) *HSQLDB*, (2) the *Geronimo* application server, and (3) the *GlassFish* application server, because each application has a scalability problem caused by lock contentions. We identified the critical sections where lock contentions frequently occur, and modified the source code so that the critical sections are executed transactionally. However, this simple modification proved insufficient to achieve reasonable performance because of excessive data conflicts. We found that most of the data conflicts were caused by application-level optimizations such as reusing objects to reduce the memory usage. After modifying the source code to disable those optimizations, the TM-based applications showed higher or competitive performance compared to lock-based applications. Another finding is that the number of variables that actually cause data conflicts is much smaller than the number of variables that can be accessed in critical sections. This implies that the performance tuning of TM-based applications may be easier than that of lock-based applications where we need to take care of all of the variables that can be accessed in the critical sections.**

I. INTRODUCTION

Transactional memory (TM) shows promise as a new concurrency control mechanism to replace lock-based synchronization. *Coarse-grained* locking, which uses a single lock to protect accesses to a large data structure, can lead to poor scalability. In contrast, *fine-grained* locking, which uses two or more locks to protect accesses to the decomposed components in a large data structure, can lead to error-prone code and complex programming. As a new programming paradigm, TM aims to simplify multi-thread programming and achieve high scalability by relying on a runtime system that provides transactions that can run concurrently as long as there is no data conflict.

Prototype TM systems have been evaluated on simple data structures, such as linked lists and red-black trees [1][2][3][4][5][6]. Recently, some TM systems have been evaluated on more complex benchmarks with various data structures, transaction sizes, and data access patterns [7][8][9][10]. However, there are few studies in which a TM system is evaluated with real applications, and the real-world benefits of and barriers to TM remain unknown.

Zyulkyarov et al. discussed their experiences when they *transactified* the *parallel Quake* game server [11] to create the *atomic Quake* game server [12]. We use the verb "transactify" to mean modifying the source code of a program to define one or more sections that are executed by using transactions on a TM system. They successfully simplified the source code of the parallel Quake game server, which was parallelized by using fine-grained locks, by transactifying all of the critical sections. However, the atomic Quake game server could not clearly demonstrate performance advantages for the software transactional memory (STM) system because of compiler bugs and excessive aborts.

In this paper, we present a detailed analysis of the behavior of real applications on a software transactional memory system that we developed. Based on this analysis, we aim to clarify the required programming work to achieve reasonable performance in TM-based applications. We also introduce some cases where existing TM programming idioms (e.g., *open nested transactions* [13]) are available. These case studies will help TM programmers write high performance programs, and provide useful insights for developers of TM systems in using TM programming idioms.

We selected three existing Java applications: (1) *HSQLDB* [14], (2) the *Geronimo* application server [16], and (3) the *GlassFish* application server [17]. Each application has a scalability problem caused by lock contentions. The ideal scenario for TM would be if such scalability problems could be resolved with limited programming work. For this scenario, we identified the critical sections where lock contentions occur frequently and modified the source code to execute those critical sections transactionally. We ran the TM versions of the applications on our prototype Java STM system.

In our experiments, the TM versions significantly degraded the performance of HSQLDB and GlassFish because of excessive data conflicts. These experimental results indicate that just defining transactions in the source code is insufficient to achieve good performance.

We investigated the causes of data conflicts and how to remove them. An interesting finding was that most of the data conflicts are caused by application-level optimizations such as reusing objects to reduce the memory usage. Since such application-level optimizations do not affect the core behavior of the applications, we were able to avoid some of the data conflicts by modifying the source code to disable

T. Nakaike, R. Odaira, and T. Nakatani are with IBM Research - Tokyo, Kanagawa-ken 242-0001 Japan (e-mail: {nakaike, odaira, nakatani}@jp.ibm.com). Maged M. Michael is with IBM Research, Yorktown Heights, NY, USA (e-mail: magedm@us.ibm.com).

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

such optimizations. After reducing data conflicts, the TM version of HSQLDB shows higher performance than the original version when the number of threads is more than four, and the TM version of GlassFish shows performance comparable to the original version which uses fine-grained locks.

The application-level optimizations that we have disabled cause no performance problems in the original applications. They cause performance problem only in TM-based applications. This may be because our TM-based applications are built from lock-based applications.

Another finding is that the number of variables that cause data conflicts is much smaller than the number of variables that can be accessed in critical sections. When we tune lock-based applications by using finer-grained locks, we need to carefully investigate how each variable is accessed in the critical sections. In contrast, when we tune TM-based applications, we can focus on the few variables where data conflicts actually occur at runtime. This indicates that the performance tuning of TM-based applications might be easier than that of lock-based applications.

The contributions of this paper are:

- Performance evaluations of TM-based applications before and after performance tuning
- Case studies of programming for high performance TM programs
- Examples of TM programming idioms

The rest of this paper is organized as follows. Section II describes our Java STM system. Section III introduces our experimental environment. Section IV shows the data for lock contentions and describes how we transactified our three selected applications. Section V presents how we modified the source code to avoid some of the data conflicts. Section VI shows our experimental results and discusses the programming of TM-based applications. Section VII reviews related work and Section VIII summarizes this paper.

II. JAVA STM IMPLEMENTATION

Fig. 1 is an overview of our Java STM system. A transaction is manually defined in the source code by putting a critical section between the `begin` method and the `end` method. The first catch block in the retry loop catches an exception that is thrown by an aborted transaction. The second catch block catches a false exception that is thrown by a transaction that encounters an inconsistent state. A runtime exception such as NullPointerException can incorrectly occur in an inconsistent state. When such an exception is caught, the transaction is validated. If the transaction is valid, then the caught exception is thrown again as a regular exception. Otherwise, the transaction is retried.

Our JIT compiler generates two versions of each method invoked in a transaction, a non-transactional version and a transactional version [18][19][20]. All of the heap accesses are replaced with barriers in the transactional version. An invoked method is compiled at its first invocation as a non-transactional version, and a transactional version is generated only when the method is invoked within a transaction. At each barrier, an STM runtime function is called to instrument each heap access.

Our Java STM system provides only *weak atomicity* without any guarantee of the safety of *publication* or *privatization* [20][21], and does not permit a monitor object to be used by synchronized blocks both inside and outside of the transaction. In the transactional version of a method, our JIT compiler replaces the operation of acquiring a lock at the entry of a synchronized block with a barrier, which sets a flag on the header of a monitor object and then checks whether or not the lock of the monitor object has been acquired. If the lock was acquired, then an exception is thrown. In the non-transactional version of a method, the flag of each monitor object is checked immediately after the lock is acquired. If the flag is set, an exception is thrown. When we find that such an exception was thrown for a synchronized block, then we need to manually replace the synchronized block with the transaction. Note that this replacement can be automated by using a technique proposed in [22].

A. STM Runtime

We use a 128-byte mapping from a shared memory location to an ownership record table entry that is used to control memory accesses and to detect data conflicts. We

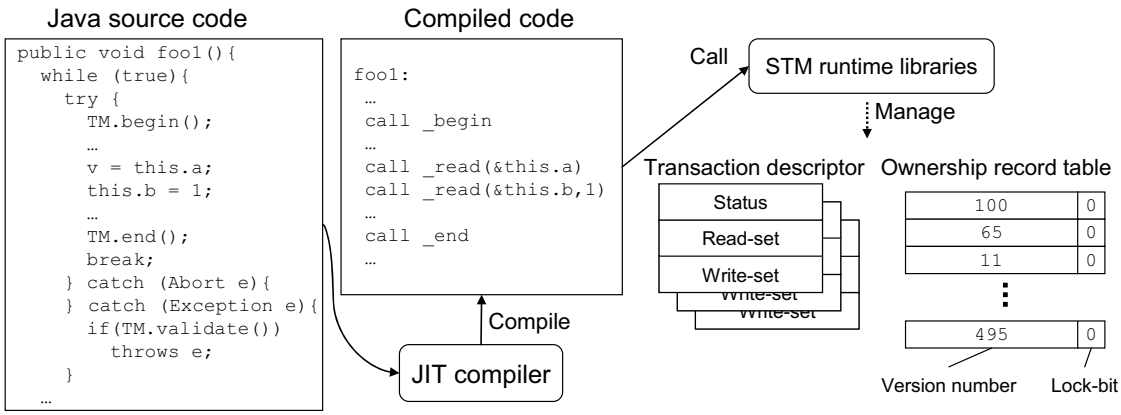


Fig. 1. Overview of the Java STM system

decided on this value based on the cache line size of the evaluation platform. Each thread has a thread-local data structure called a *transaction descriptor* with a *read-set* and a *write-set*. A read-set entry is a pair of an address and a version number. Since we use *lazy-versioning* [2][19][23], a write-set entry buffers a value to be written in a transaction. Using lazy versioning, a transactional read needs to check if the read location was already written by the transaction, and if so, search for the latest value in the write-set. To reduce the search cost, our STM runtime uses a *bloom filter* [24] and hashes the write-set by using the lower bits of the address [25].

To avoid the writes to shared metadata by non-conflicting transactions, we used a read-set validation mechanism for a policy of invisible reads instead of using global timestamps [2][26][27]. To avoid restrictions and overhead on access to transaction descriptors, we do not require the STM operations to be non-blocking, but rather use locking in the metadata operations.

Our STM runtime supports an irrevocable mode to avoid starvation in which a long-running transaction is repeatedly aborted by short running transactions. Each transaction counts its number of aborts. If the number of aborts reaches a threshold, the transaction tries to acquire a lock to switch to an irrevocable mode. If a transaction acquires this lock, then the other transactions cannot proceed to the commit process.

B. JVM and JIT Compiler Support for Transactional Execution

Our JVM aborts all of the transactions before each Garbage Collection (GC) because GC can invalidate the metadata associated with addresses by moving the objects. Our JVM periodically validates transactions by using a validation thread and an asynchronous event checking mechanism. The validation thread periodically wakes up, and then sends a validation event to each running thread. Each thread checks the existence of the event at an asynchronous checkpoint. A transaction running on a thread thus receives a validation event and then validates its own read-set. Since asynchronous checkpoints are placed on the back edges in the control-flow graph, an infinite loop caused by an inconsistent read is avoided. Note that there is no overhead on the execution path where no asynchronous event occurs, because asynchronous checkpoints are inserted even without our read-set validation mechanism for other purposes, such as GC.

We have implemented two well-known compiler optimizations [18][28]. One is the inlining of barriers. Our JIT compiler inlines the fast paths of the read barriers. This reduces the overhead for calling STM runtime functions. The second optimization is the elimination of the barriers for *transaction-local objects* that are created in a transaction. When our JIT compiler finds a statement creating an object, it eliminates the barriers for accesses to the object only if the object does not escape outside of the compilation scope. We also support a runtime filter for transaction-local objects. We prepare a transaction-local flag in each object header. This flag is set when an object is created in a transaction. Our JIT compiler adds code to check the flag and skip the execution of the barrier.

We also implemented read-barrier elimination for read-only variables. This optimization is similar to the optimization proposed in [29] to accelerate transactions in a weakly atomic STM system. In our read-barrier elimination, the number of writes for each variable is counted at runtime by scanning every method at its first invocation in a transaction. If no new method is invoked in the transactions over a certain period, the method is recompiled (upgraded) to eliminate the read- barriers. When a write to a variable appears in a newly invoked method after the read-barriers have been eliminated, we recover the read-barriers, and then abort all of the transactions before executing the new write.

III. Experimental Environment

Here are the applications we selected for testing:

- **HSQLDB**: This is a relational database written in Java [14]. Our workload is the banking scenario in the *DaCapo* benchmark [15].
- **Geronimo application server**: This is an application server implementation [16] provided by Apache. Our workload is *DayTrader 2.0* [30], which is a 3-tier (client–server–database) application that emulates an online stock trading system.
- **GlassFish application server**: This is an application server implementation [17] provided by Oracle. Our workload is *PetStore 2.0* [31], which is a 3-tier application that emulates an online pet shop.

We used a 4.7-GHz 32-way Power6 processor for the performance measurements. In addition, we used two machines to run DayTrader 2.0 on the Geronimo application server and to run PetStore 2.0 on the GlassFish application server. Fig. 2 shows the experimental environment. We changed the number of the active cores of the Power6 machine to measure the scalability instead of changing the number of software (Java) threads, because it is difficult to control the number of software threads in an application

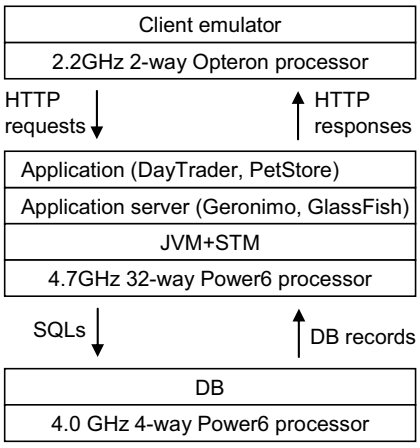


Fig. 2. Experimental environment

server. The client emulator runs on a 2-way Opteron processor. The client emulator sends HTTP requests to the application server and receives the HTTP responses. The application server processes each request by accessing the database, which ran on a 4-way Power6 processor, and returns the HTTP responses to the client emulator.

IV. Transactifying Applications

In our three selected applications, we did not define transactions for all of the critical sections because an excessive use of transactions in critical sections where lock contentions rarely occur can unnecessarily increase the instrumentation overhead. We analyzed which critical sections are the sources of most lock contentions, and defined transactions only for those critical sections.

Fig. 3 shows the breakdown of lock contentions. In the IBM JVM that we used for our experiments, lock contentions are resolved by using a *three-tier locking* scheme [32] where a thread waits for a certain time for a lock to be released using a spin-wait loop. We used *Java Lock Monitor (JLM)* [33] to measure the number of spins as the number of lock contentions.

In the remainder of this section, we describe the causes of lock contentions and how we defined transactions in each application.

A. HSQLDB

As shown in Fig. 3, most of the lock contentions occur on the `Database` object which is used in a single synchronized block as a monitor object. This synchronized block executes an SQL statement, and thus fully serializes the execution of the SQL statements. We defined a transaction instead of the synchronized block to allow the concurrent execution of SQL statements.

B. Geronimo Application Server

As shown in Fig. 3, 95% of the lock contentions occur on a `ReentrantLock` object, which is a read-write lock (not the `ReentrantLock` provided by the Java concurrent package). This lock object is used to access the *OpenJPA* [34] cache. OpenJPA is one of the modules of the Geronimo application server, and it is an implementation of the J*ava Persistence API (JPA)* [35], which defines an interface to manage persistent objects mapped to database records. OpenJPA has a cache to store persistent objects. If a persistent object is in the cache, we can avoid a database access.

The read-write lock is implemented by using synchronized blocks. A read lock is acquired by incrementing the number of readers in a synchronized block, and it is released by decrementing the number of readers in another synchronized block. A write lock is acquired by writing the thread ID to the read-write lock object in a synchronized block.

Although many lock contentions occur on the synchronized blocks that perform the operations for lock acquisition and release, those operations are not targets for transactional execution. The targets for transactional execution are cache accesses. Therefore, we defined transactions for the critical sections that are protected by read-write locks.

The OpenJPA cache has three maps: (1) a pinned map, (2) a cached map, and (3) a soft reference map. Fig. 4 shows the `get` method of the class that implements the OpenJPA cache. The pinned map stores the objects specified by an application developer. The cached map is the main storage of the cache. Since the number of objects stored in the cached map is limited, an object is moved to the soft reference map when the number of stored objects exceeds a threshold. Since objects stored in the soft reference map are weakly referenced, they are removed at GC time.

C. GlassFish Application Server

As shown in Fig. 3, 45% of the lock contentions occur on multiple `ConcurrencyManager` objects which are read-write locks implemented by using synchronized blocks. This lock is used to access the *TopLink Essentials* [36] cache. The GlassFish application server includes a module that implements JPA in a way similar to the Geronimo application server. The JPA implementation in the GlassFish application server is called TopLink Essentials, which also has a cache to store persistent objects. We defined transactions for cache accesses. These transactions cover 18% of lock contentions that occur on an `IdentityMapManager` object, because that object is accessed in the critical sections to access to the cache. Unfortunately, we could not use transactions for the critical sections protected by a `PortUnificationPipe-`

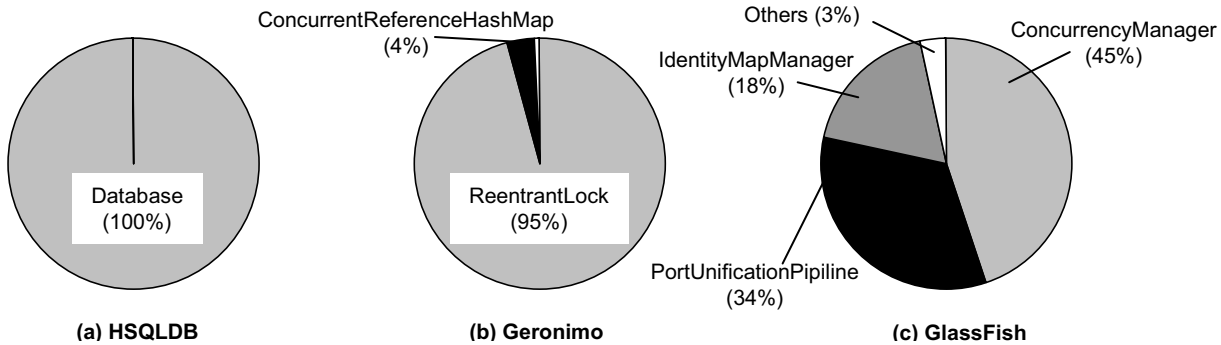


Fig. 3. Breakdown of lock contentions by class names

line object, because those critical sections include operations to control threads, such as thread creation.

TopLink Essentials uses fine-grained locks for cache accesses, unlike OpenJPA which uses a single read-write lock. Each cached object has its own key and each key has its own read-write lock. Therefore, the data shown in Fig. 3 means that lock contentions frequently occur on a fine-grained lock when multiple threads concurrently try to access a single cached object.

Fig. 5 is a flowchart for a typical read access to the cache. First, we try to find a cache key in a hash table based on a given primary key. We call this hash table the cache key table. If a cache key is found, the read lock of the cache key is acquired, and then the persistent object is read from the cache. Otherwise, the record is read from the database. After that, a synchronized block that uses the cache key table as a monitor executes three operations: (1) creating a cache key based on the primary key, (2) adding the cache key to the cache key table, and (3) acquiring the lock for the cache key. The synchronization on the cache key table is needed to avoid creating multiple cache keys for a single primary key. After a lock for a cache key is acquired, the persistent object is created based on the record and put into the cache. Finally, the lock of the cache key is released. While a persistent object is being created, the lock of the cache key is acquired to avoid creating multiple persistent objects for a single cache key. When a persistent object is read, the lock of the cache key is acquired to avoid reading the object while the persistent object is being created.

```
public Object get(Object key) {
   readLock();
   Object val = pinnedMap.get(key);
   if (val != null) return val;
   val = cacheMap.get(key);
   if (val == null) {
     val = softMap.get(key);
     if(val != null) put(key, val);
   }
   readUnlock();
   return val;
}
```

Fig. 4. Read access to the OpenJPA cache

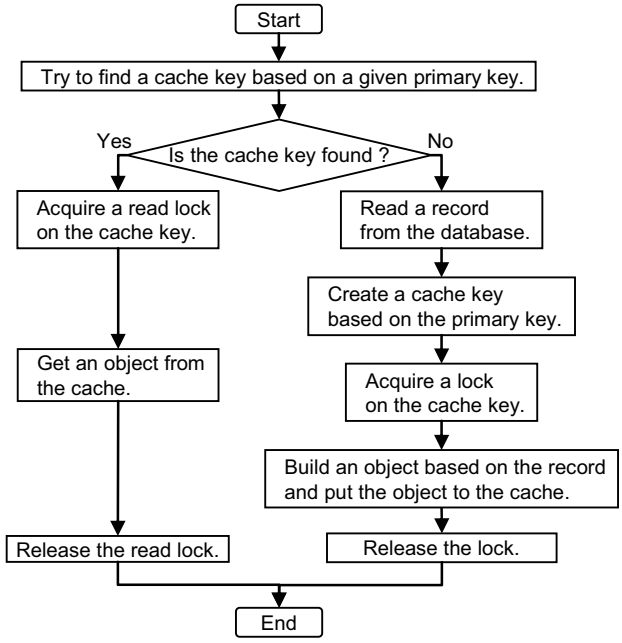


Fig. 5. Read access to the TopLink Essentials cache

V. Reduction in Data Conflicts

Our full experimental results appear in Section VI, but to summarize them briefly, the TM versions ran significantly slower than the original versions of HSQLDB and the GlassFish application server because of excessive data conflicts. We developed a tool to identify the conflicting variables and investigated the causes of these data conflicts. In this section, we describe the causes of the data conflicts and our solutions to address some of them.

We found that most of the data conflicts were caused by application-level optimizations, such as reusing objects to reduce memory usage. Application-level optimizations do not affect the core behavior of the application so we could disable these application-level optimizations to reduce the data conflicts. In Section VI, we show how the performance was affected by disabling these application-level optimizations.

In this section, we also discuss how we could reduce the data conflicts or reduce the overhead for rollbacks while retaining the application-level optimizations if our STM system supported three TM programming idioms: (1) *open nested transactions (ONTs)* [13], (2) *abstract nested transactions (ANTs)* [37], and (3) *early release (ER)* [38]. ONTs allow a nested transaction to be committed before the outer transaction is committed. ANTs allow a nested transaction to be re-executed at the end of the outermost transaction as long as data conflicts are limited to the variables that were accessed inside of the nested transactions and the values written in the nested transactions were not accessed outside of the nested transactions. ER allows some transaction logs (i.e. read-set entries or write-set entries) to be excluded from data conflict detection at commit time.

A. Problem 1 in HSQLDB

The first cause of data conflicts in HSQLDB was a global counter tracking the number of database records. This global counter is used to deliberately invoke GC by calling `java.lang.System.gc()` when the number of created records exceeds a threshold. Since the global counter is incremented for each creation of a database record, data conflicts occur frequently for the global counter.

We solved this problem by simply eliminating the deliberate GC and allowing the JVM to invoke GC as needed. Alternatively, ANTs could be used to control the GC if the TM system supports `java.lang.System.gc()` within a transaction. The global counter would be accessed only when its value is incremented and the value would be checked to determine if GC was needed. Enclosing the increment of the

value and the check of the value in two ANTs would allow re-execution of the ANTs at the end of a transaction, and could reduce the overhead for rollbacks.

B. Problem 2 in HSQLDB

The second cause of data conflicts in HSQLDB was a global counter for the number of the primitive objects pooled in a hash table, used to avoid recreating the same primitive objects. This object pool returns an existing primitive object for a given primitive value if the object exists in the hash table. Otherwise, the object pool creates a new object, puts the object into the hash table, and returns the object. Since the global counter is incremented in each creation of a new primitive object, data conflicts frequently occur for the global counter.

We solved this problem by simply removing the object pooling, even though this can increase the memory usage. Alternatively, ONTs could be used to retain the object pooling. Enclosing all of the operations related to the object pooling in ONTs could reduce the overhead for rollbacks by limiting the scope of the re-execution caused by a rollback to the operations related to the object pooling. This does not affect the core behavior of HSQLDB, because an object returned from the object pool is immutable and its content is accessed consistently outside of the ONTs. However, there is a risk in adding unnecessary objects into the object pool when an outer transaction is aborted. Using ANTs can cause problems because a value written in an ANT to create a new primitive object might be accessed outside of the ANT.

C. Problem 3 in HSQLDB

The third cause of data conflicts in HSQLDB was objects that express SQL statements. When an SQL statement is given, it is compiled, and the compiled SQL statement is stored in an object. This object is retained during the execution of the SQL statement and can be shared among multiple threads. Although this avoids creating an object for each execution of the SQL statement, data conflicts occur frequently for an object that involves updates of the instance fields in each execution of the SQL statement.

We solved this problem by using a thread-local object to express the compiled SQL statement, even though this can increase the memory usage as the numbers of threads increase. Alternatively, ER could be used to avoid creating the thread-local objects. An object to express a compiled SQL statement is used for transaction-local storage. In other words, the values written to the object within a transaction are never read by the other transactions. ER would allow buffering the writes to the object until the end of the transaction while excluding those writes from the commit.

D. Problem 4 in HSQLDB

The last cause of data conflicts in HSQLDB was a global counter used to generate a unique ID for each database record. This was the only problem related to the core behavior of HSQLDB, unlike the other problems we found. Since the global counter is incremented in each creation of a database record, data conflicts occur frequently on this global counter.

We solved this problem by modifying our Java STM system to allow an atomic update of this global counter directly in the shared memory within the transaction. This is similar to ONTs, and thus ONTs could be used for reducing the rollback overhead in these situations.

E. Problem 1 in GlassFish

The first cause of data conflicts in GlassFish was a shared variable for a most-recently-accessed hash table. The TopLink Essentials cache consists of multiple hash tables. Each hash table only stores the persistent objects of a specific class. These hash tables are stored in another hash table, and indexed by the class of the persistent objects to be stored. Therefore, there is an indirection to get each persistent object. To avoid this indirection, the most-recently accessed hash table is kept in a shared variable. When a hash table is retrieved from the first hash table, its reference is written to the shared variable. If a thread finds its target hash table in the shared variable, then it can avoid the indirection. However, since there are frequent writes to the shared variable, there are frequent data conflicts over it.

We solved this problem by eliminating this optimization, though this causes every access to a persistent object to require indirection. Alternatively, ONTs could be used to enclose the operations related to hash table access from the first hash table. However, using ONTs might not improve the performance, because the cost for rolling back the ONTs could be larger than the cost to access the first hash table.

F. Problem 2 in GlassFish

The second cause of data conflicts in GlassFish is an object that is used to compute the hash code. As mentioned in Section IV.C, to retrieve a persistent object from the cache we need to get a cache key from the cache key table based on the given primary key. Since each cache key is indexed by its own hash code, a new cache key needs to be created to find an existing cache key in the cache key table. To avoid creating a cache key in each search for an existing cache key, a cache key is stored in a shared variable. This cache key is only used to compute the hash code based on the given primary key. Since the hash code field of the cache key is updated in each search for a cache key, data conflicts occur frequently.

We solved this problem by creating a cache key in each search for an existing cache key, even though this can increase the memory usage. This pattern is similar to Problem 3 in HSQLDB. The cache key stored in the shared variable is used for transaction-local storage. Therefore, ER could be used to reduce these data conflicts.

G. Problem 3 in GlassFish

The last cause of data conflicts in GlassFish was the LRU (Least Recently Used) list for controlling the cache. The TopLink Essentials cache evicts the LRU cached object. When a cached object is accessed, it is placed at the head of

the list. When the number of the cached objects exceeds a threshold, the object currently at the end of the list is evicted from the cache. Since the head of the list is updated on each access to the cache, data conflicts occur frequently.

We solved this problem by not using LRU, even though this can reduce the efficiency of the cache. Each new object is placed at the end of the list. When the number of the cached objects exceeds a threshold, the object at the head of the list is evicted. With this modification, data conflicts never occur as long as a new object is not added to the list and a cached object is not evicted. Alternatively, ANTs could be used to enclose all of the operations related to the LRU control, because the content of each cached object is not accessed inside of the ANTs though it is accessed outside of the ANTs.

VI. Performance Evaluation

A. Experimental Results

We implemented our Java STM system on a 64-bit JVM. We used 2 GB of heap in our experiments. The interval for periodic read-set validation was 1 millisecond. Although we tested longer intervals such as 10 milliseconds, we did not see any changes in the performance.

We compared four versions of each application.

- **Lock**: This is the original version using locks.
- **DowngradedLock**: This is downgraded from the original version by disabling some of the application-level optimizations mentioned in Section V.
- **TM**: This is transactified from the original version as described in Section IV.
- **TunedTM**: This is tuned from the TM version to reduce the data conflicts described in Section V.

For Geronimo, there is no data for DowngradedLock and TunedTM because tuning was not done.

Table 1 shows the basic characteristics of each application. The runtime statistics such as transaction coverage were measured for a single thread.

Fig. 6 shows the speed-up, which is for the normalized throughput based on the throughput of the original version for a single thread. Fig. 7 shows the abort ratios of the TM versions and the tuned TM versions. In all of the applications, the TM version and the tuned TM version showed lower throughput than the original version for a single thread because of the instrumentation overhead.

The TM versions of HSQLDB and GlassFish showed poor performance because of excessive aborts, even when we increased the number of processors. With 32 processors, the TM versions of HSQLDB and GlassFish had 70% and 45% abort ratios respectively. In contrast, the TM version of Geronimo shows higher throughput than the original version when the number of processors is greater than 8. This is because the Geronimo application server had smaller transaction coverage than the other two applications as shown in Table I. This small transaction coverage minimizes the instrumentation overhead and the wasted time for aborted transactions.

In HSQLDB, the tuned TM version showed higher throughput than the original version when the number of processors is greater than 2. In the GlassFish application server, the tuned TM version significantly improved the scalability compared to the TM version, though it did not outperform the original version. Note that the performance of the tuned TM version approached that of the original version when we increased the number of processors from 8 to 16. The tuned TM version of GlassFish showed 18% performance degradation compared to the original version for more than 8 processors while it showed 24% performance degradation for fewer than 16 processors.

In HSQLDB and the GlassFish application server, the downgraded version (DowngradedLock) showed similar throughput compared to the original version. Therefore, the application-level optimizations for HSQLDB and the GlassFish application server were not effective for our test workloads. Since some of the application-level optimizations should reduce the memory usage, we measured the frequency of GCs by changing the heap size. Fig. 8 shows the frequency of GCs per million operations (i.e. the processed database transactions in HSQLDB and the processed HTTP requests in the GlassFish application server). In HSQLDB, we did not see any major differences in the GC frequencies between the original version and the downgraded version. In the GlassFish application server, we saw a slight difference between the original version and the downgraded version, but the overall performance was not affected. As shown in Fig. 8, the TM version and the tuned TM version showed higher frequencies of GC than the original version. This is caused by the objects that are created unnecessarily in the aborted transactions.

Table I. Transaction statistics

Application	Transaction coverage (%)	Read barriers		Write barriers		Read-set size		Write-set size	
		Static	Dynamic	Static	Dynamic	Average	Max.	Average	Max.
HSQLDB	92	446	155	224	31	87	334	25	120
Geronimo	0.5	1611	115	319	7	40	9888	2	272
GlassFish	13	907	380	133	4.6	90	52249	3.9	3617

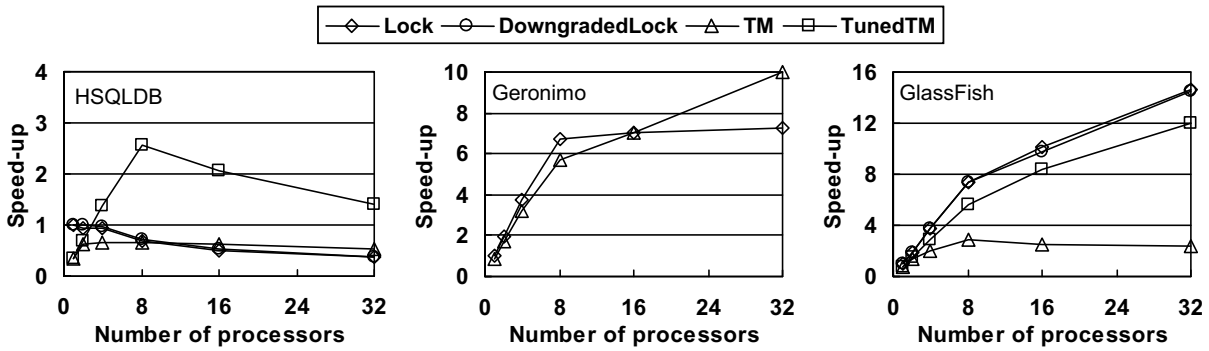


Fig. 7. Speed-up which is a normalized throughput based on the throughput of the original version for a single thread (there are no data for DowngradedLock and Tuned TM in Geronimo because tunings are not performed in Geronimo)

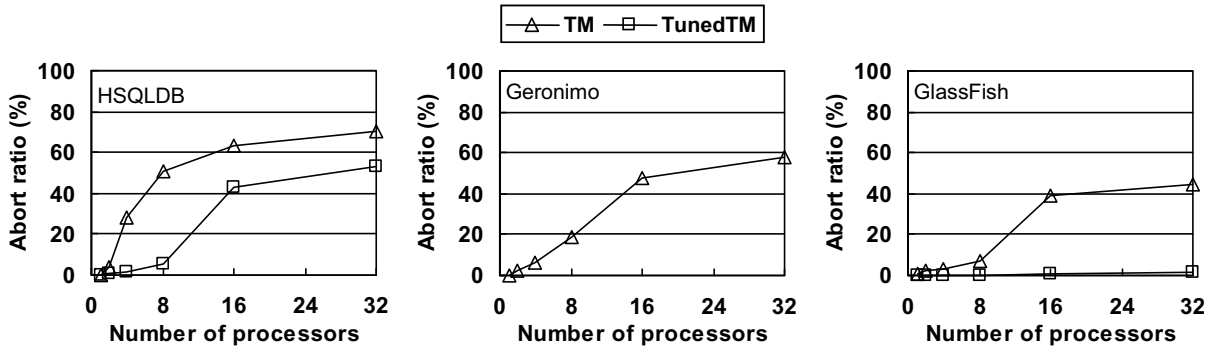


Fig. 6. Abort ratios (there is no data for TunedTM in Geronimo because tuning was not done for Geronimo)

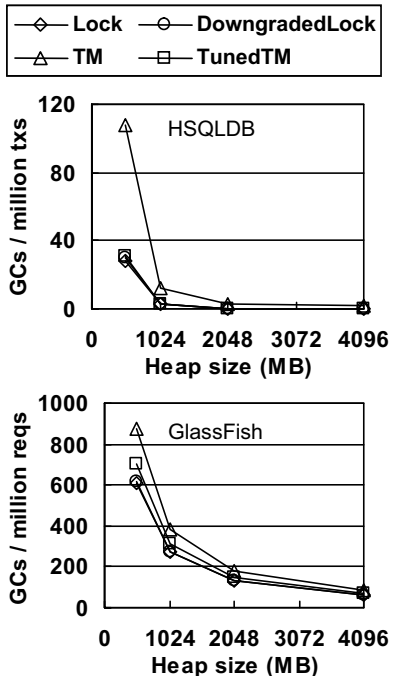


Fig. 8. GC frequency

B. Assessment of Programming Efforts

As shown in Section VI.A, we were able to improve the performance of the TM-based applications by working to reduce the data conflicts. In this section, we discuss whether the programming effort required to improve the performance of lock-based applications by using TM is smaller than the programming effort required to improve the performance without using TM.

Here are the development steps that we used to create the TM-based applications from the existing applications and to improve the performance.

Step 1. Identify where lock contentions frequently occur.
Step 2. Apply the transactional execution to the critical sections identified in Step 1.
Step 3. Measure the performance. If the performance is unsatisfactory and there are no major data conflicts, then return to Step 1.
Step 4. Analyze the causes of the data conflicts (as in HSQLDB and the GlassFish application server).
Step 5. Modify the source code to reduce the data conflicts.
Step 6. Measure the performance. If the performance is satisfactory, then the work is finished. If there are still major data conflicts, then return to Step 4. If the performance is not satisfactory and there are no major data conflicts, but there are still major lock contentions, then return to Step 1.

The analysis of data conflicts is done step-by-step. We will first find the major data conflicts on one variable. After removing that variable with its conflicts, we look for the next conflicting variable.

Here is a typical development cycle to improve the performance of a lock-based application without using TM.

Step 1. Identify where lock contentions frequently occur.
Step 2. Identify all of the variables that can be accessed in the critical sections identified in Step 1.
Step 3. Categorize the variables identified in Step 2 into shared variables and thread-local variables.
Step 4. Modify the source code to convert each shared variable into a thread-local variable if possible and use fine-grained locks.
Step 5. Measure the performance. If the performance is still unsatisfactory, then return to Step 1.

In the work to reduce data conflicts for TM-based applications and the work to use fine-grained locks for the lock-based applications, we often must investigate how each shared variable is accessed. The main difference between TM-based programming and lock-based programming involves the number of variables that must be examined to improve the performance. In lock-based applications, we need to study all of the variables that can be accessed in critical sections, which may be a large number of variables. As shown in Table I, the static number of barriers that are inserted to instrument heap accesses is more than a hundred, and this represents roughly the number of variables that can be accessed in critical sections. In contrast, we can focus on the variables where data conflicts actually occurred at runtime in TM-based applications. That number may be much smaller than the number of variables that can be accessed in critical sections. In our actual experiments, we were able to focus on just four variables in HSQLDB and three variables in the GlassFish application server.

VII. Related Work

When TM research first started, simple data structures such as red-black trees and linked-lists were used as benchmarks [1][2][3][4][5][6]. Transactional memory coherence and consistency (TCC), which is one of the hardware transactional memory (HTM) systems, was evaluated with existing benchmarks for C/C++ or Java such as SPLASH2 or JavaGrande [39][40][41][42]. Recently, benchmarks with various data structures, transaction sizes, and data access patterns have been developed to evaluate TM systems [7][8][9][10]. Since the main purposes of these benchmarks are: (1) to enhance a TM system by testing various configurations, (2) to compare different TM implementations, and (3) to understand the hardware resources required for an HTM system, it is difficult to assess the real-world benefits of or barriers to TM.

Zyulkyarov et al. transactified the parallel Quake game server [11] into the atomic Quake game server [12]. They simplified the source code by replacing the critical sections guarded by fine-grained locks with transactions. The main obstacle in the development of the atomic Quake game server was the non-block-structured critical section. We had similar experiences. However, the atomic Quake game server was significantly less scalable compared to the parallel Quake game server. In contrast, we found that TM-based applications can have higher or competitive performance compared to lock-based applications.

There have been several attempts to improve transactified code by modifying the source code. Yoo et al. [43] introduced a new pragma called *tm_waiver* in a C/C++ STM system to specify which functions need no instrumentation. Although tm_waiver was effective in reducing the instrumentation overhead and false conflicts, it cannot be used to eliminate the real conflicts that we found in HSQLDB and the Glassfish application server. Damron et al. evaluated their Hybrid TM system [44] on the Berkeley DB and SPLASH-2 benchmarks, and they found real conflicts on some variables such as a global counter used to generate a unique ID. Those real conflicts are caused by frequently updating the same variable, and similar cases exist in HSQLDB and the Glassfish application server.

VIII. Conclusion

In this paper, we described our experiences building TM-based applications from existing applications and tried to clarify what programming work is needed to achieve reasonable performance. We added transactional execution to three applications with scalability problems. Although the performance was improved in one of the applications, it was degraded in the other two applications because of excessive data conflicts. By analyzing the data conflicts, we found that most of the problems were caused by application-level optimizations. By disabling those optimizations, the TM versions of the applications were able to achieve higher or competitive performance compared to the original versions. Another finding is that most of the data conflicts are caused by a small number of variables compared to the total number of variables that can be accessed in critical sections. This is evidence that the programming effort required in TM-based applications is more limited than the programming work required in lock-based applications.

References

[1] B. Saha, A. Adl-Tabatabai, R.L. Hudson, C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187-197, March 2006.

[2] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194-208, September 2006.

[3] M. Herlihy and J.E.B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 389-300, May 1993.

[4] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 92-101, July 2003.

[5] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204-213, August 1995.

[6] V.J. Marathe, et al. Lowering the Overhead of Software Transactional Memory. *ACM SIGPLAN Workshop on Transactional Computing*, June 2006.

[7] C. Cao Minh, J. Chung, C. Kozyrakis, and O. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the 11th IEEE International Symposium on Workload Characterization*, September 2008.

[8] F. Zyulkyarov, et al. WormBench – A Configurable Workload for Evaluating Transactional Memory Systems. In *Proceedings of 2008 Workshop on Memory Performance: Dealing with Applications, Systems, and Architecture*, pages 61-68, October 2008.

[9] M. Ansari, C. Koteselidis, K. Jarvis, M. Lujan, C. Kirkham, and Ian Watson. Expericences using Adaptive Concurrency in Transactional Memory with Lee's Routing Algorithm. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 261-262, February 2008.

[10] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the Second European Systems Conference EuroSys*, 2007.

[11] A. Abdelkhalek and A. Bilas. Parallelization and Performance of Interactive Multiplayer Game Servers. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium,* pages 72-81, April 2004.

[12] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25-34, February 2009.

[13] Y. Ni, et al. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68-78, 2007.

[14] hsqldb – 100% Java Database. http://hsqldb.org/.

[15] S.M. Blackburn, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169-190, October, 2006.

[16] Apache Geronimo. http://cwiki.apache.org/geronimo/.

[17] GlassFish – Open Source Application Server. https://glassfish.dev.java.net/.

[18] A. Adl-Tabatabai, B.T. Lewis, V. Menon, B.R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26-37, June 2006.

[19] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 388-402, October 2003.

[20] T. Shpeisman, et al. Enforcing isolation and ordering in STM. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78-88, June 2007.

[21] V. Menon, S. Balensiefer, T. Shpeisman, A. Adl-Tabatabai, R.L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java stm. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 314-325, June 2008.

[22] L. Ziarek, A. Welc, A. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A Uniform Transactional Execution for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, July 2008.

[23] V.J. Marathe, et al. Lowering the Overhead of Software Transactional Memory. *ACM SIGPLAN Workshop on Transactional Computing*, June 2006.

[24] B.H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of ACM*, 13(7), pages 422-426, 1970.

[25] M.F. Spear, L. Dalessandro, V.J. Marathe, and M.L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Computing*, pages 141-150, February 2009.

[26] M.F. Spear, V.J. Marathe, W.N. Scherer III, and M.L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of 20th International Symposium on Distributed Computing*, pages 179-193, September 2006.

[27] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of 20th International Symposium on Distributed Computing*, pages 284-298, September 2006.

[28] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14-25, June 2006.

[29] N.G. Bronson, C. Kozyrakis, and K. Oluotun. Feedback-Directed Barrier Optimization in a Strongly Isolated STM. In *Proceedings of the 36th ACM SIGPLAN – SIGACT Symposium on Principles of Programming Language*, pages 213-225, January 2009.

[30] Apache DayTrader Benchmark Sample. http://cwiki.apache.org/GMOxDOC20/daytrader.html.

[31] Java™ Pet Store Demo. https://blueprints.dev.java.net/petstore/

[32] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.

[33] JLM – Java Lock Monitor. http://perfinsp.sourceforge.net/jlm.html

[34] Apache OpenJPA. http://openjpa.apache.org/.

[35] JSR220: Enterprise JavaBeansTM, Version 3.0. Java Persistence API. 2006.

[36] TopLink JPA. http://www.oracle.com/technology/products/ias/toplink/jpa/index.html

[37] T. Harris and S. Stipic. Abstract nested transactions. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'07)*, 2007.

[38] K. Fraser. Practical Lock Freedom. PhD thesis, Computer Laboratory, University Cambridge, 2003.

[39] B.D. Carlstrom, et al. Transactional Execution of Java Programs. *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

[40] J. Chung, C. . Minh, B D. Carlstrom, and C. Kozyrakis. Parallelizing SPECjbb2000 with Transactional Memory. *Workshop on Transactional Memory Workloads*, 2006.

[41] J. Chung, et al. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February, 2006.

[42] L. Hammond, et al. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[43] R.M. Yoo, Y. Ni, A. Welc, B. Saha, A. Adl-Tabatabai, and H. S. Lee. Kicking the Ties of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.

[44] P. Darmron, A. Fedorova, Y. Lev, V. Luchangco, M. Moior, and D. Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336-346, October 2006.

EigenBench: A Simple Exploration Tool for Orthogonal TM Characteristics

Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun
Pervasive Parallelism Laboratory, Stanford University
{hongsup, tayo, jared, nbronson, kozyraki, kunle}@stanford.edu

***Abstract*—There are a significant number of Transactional Memory(TM) proposals, varying in almost all aspects of the design space. Although several transactional benchmarks have been suggested, a simple, yet thorough, evaluation framework is still needed to completely characterize a TM system and allow for comparison among the various proposals. Unfortunately, TM system evaluation is difficult because the application characteristics which affect performance are often difficult to isolate from each other. We propose a set of orthogonal application characteristics that form a basis for transactional behavior and are useful in fully understanding the performance of a TM system.**

In this paper, we present EigenBench, a lightweight yet powerful microbenchmark for fully evaluating a transactional memory system. We show that EigenBench is useful for thoroughly exploring the orthogonal space of TM application characteristics. Because of its flexibility, our microbenchmark is also capable of reproducing a representative set of TM performance pathologies. In this paper, we use Eigenbench to evaluate two well-known TM systems and provide significant insight about their strengths and weaknesses. We also demonstrate how EigenBench can be used to mimic the evaluation coverage of a popular TM benchmark suite called STAMP.

I. Introduction

Since the onset of transactional memory (TM) research, there has been an explosion in the number and variety of proposed TM systems. There are many design choices for TM, including how much functionality to put in hardware, types of conflict detection and resolution, choices for version management, overflow handling techniques, and many more [1]. The design space is now varied enough that evaluation of TM systems is an independent and interesting problem.

Attributes of a TM system affect the performance of a transactional application in different ways depending on the characteristics of the application. For example, applications with many shared transactional accesses may behave quite differently than applications with few. Application-based benchmarks are important since their patterns attempt to represent realistic workloads, but they have a limited ability to isolate the effect of each characteristic on the overall performance. For example, an application's working-set size is often tied to the size of its transactions, but these two characteristics may be completely orthogonal in terms of how they affect system performance. Current evaluation frameworks do not fully account for this reality.

In our approach, we first propose a set of orthogonal characteristics of TM applications, which we call *eigen-characteristics*. We show how together these characteristics form a basis for all TM applications in the same way that a basis in linear algebra spans a vector space. Ideal TM evaluation requires the ability to decouple the eigen-characteristics from each other and vary them independently, a methodology which we call *orthogonal analysis*. To this end, we have developed *EigenBench*, a simple yet thorough microbenchmark for fully evaluating a transactional memory system (Section II). EigenBench provides insight into a TM system's performance by fully exploring the eigen-characteristics, evaluating corners of the application space not easily reached by existing benchmarks (Section III).

In addition to application characteristics, it is useful to understand an implementation's susceptibility to *pathologies*, as they have been shown to significantly affect performance [2]. We show that EigenBench can easily reproduce a representative set of pathological cases (Section IV).

Finally, we show that the performance of TM applications can be easily explained in terms of their eigen-characteristics. We demonstrate how given these characteristics, EigenBench can closely approximate the execution behavior of that application for the purposes of evaluating a TM system. We then perform examples of this mimicry using benchmarks from the STAMP TM suite [3] (Section V).

A. Related Work

Other researchers have released TM microbenchmarks and benchmark suites. TM microbenchmarks are usually structured as a parameterizable set of transactions that operate on a shared data structure like a red-black tree or a hash table [4]–[8]. These tools are small, portable, and useful. However, they do not aim to parameterize across the entire TM design space and have not been shown to reproduce all applicable, known pathologies.

The STAMP benchmark suite [3] consists of eight configurable applications and exercises a wide range of transactional behaviors. Although STAMP successfully provides a representative set of TM applications, each application's transactional characteristics are strongly tied to its parameters and are thus difficult to decouple from each other. Being a full benchmark suite, STAMP is also more complex and requires more configuration before being used. In addition, STAMP has not been shown to exercise all pathological cases. We will demonstrate that EigenBench is flexible enough to mimic the transactional characteristics of the STAMP benchmark suite.

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

Other TM benchmark suites have been proposed, including those from Kestor et al. [9], Ansari et al. [10], and Guerraoui et al. [11]. These suites focus on particular application domains and are not designed to exercise the full breadth of TM behavior. Our approach is most like STMBench7 [11] in that we provide a single parameterized microbenchmark that can be easily configured to mimic a wide variety of workloads. However, our approach aims to be much more general and cover a larger portion of the TM application space by avoiding any particular underlying data structure. As an example, SwissTM outperformed TL2 by 3x in all STMBench7 results but only by 1.1 – 1.9x in STAMP results [12], which shows that STMBench7 did not produce performance behaviors of STAMP applications

Other researchers have also defined a set of characteristics to analyze and reproduce behaviors of existing benchmarks [13], [14]. However, their selection of characteristics were not orthogonal; they instead relied on principle component analysis (PCA) for their choices. Thus, orthogonal analysis such as what we present in Section III is not possible with those characteristics.

The specific contributions of this paper are:

- We define a set of orthogonal characteristics of TM applications, and show that an orthogonal analysis along these characteristics is extremely useful in understanding a TM system's behavior.
- We present EigenBench, a lightweight yet thorough microbenchmark designed for such an orthogonal analysis.
- We show that one can explain a TM application's performance given its eigen-characteristics and demonstrate how EigenBench can be used to approximate the behavior of real applications.
- We show that EigenBench can easily reproduce all of the pathological cases delineated by Bobba et al. [2] for the evaluated TMs.

II. EigenBench's Orthogonal Characteristics

In this section, we first define the eigen-characteristics: the set of orthogonal attributes that characterize TM applications. We then present EigenBench, a microbenchmark designed to orthogonally explore these attributes. Finally, we describe how to systematically vary those characteristics using the EigenBench parameters.

A. Eigen-Characteristics

It is well understood that applications having different characteristics vary in behavior within a single TM system [3], [13], [15]. However, there is no consensus on a standard set of characteristics by which to describe TM applications. Previous proposals have used sets of characteristics that are far from orthogonal; one may be strongly determined by others. For example, a set composed of *transaction size*, *read set size*, and *write set size* do not form an orthogonal basis, since the first depends on the others.

We propose *eigen-characteristics*, a set of orthogonal characteristics that can form a basis for all TM applications. Table I presents the eight characteristics: *concurrency, working-set size, transaction length, pollution, temporal locality, contention, predominance*, and *density*. While there can be alternative selections of orthogonal characteristics, our experience found this specific choice was intuitive and very useful in practice. By our definition, *predominance* presents the ratio of shared reads and writes to the entire application including all non-shared instructions, while *density* dictates, as a complementary measure, what fraction of the non-shared instructions are executed inside transactions.

A conventional (non-orthogonal) characteristic can now be expressed as a combination of eigen-characteristics. For example, read set size is a function of *transaction length*, *pollution*, and *temporal locality*. A small read set size can be obtained by having a small number of shared accesses, a small portion of reads among many shared accesses, or many repeated accesses to a few addresses. TM systems might vary substantially in how they handle these three cases.

Also, we are deliberate in our choice of adjectives used to describe the characteristics in Table I. We especially avoided the terms 'big' and 'small' since they are too vague: a 'big' transaction could mean a *long* one (having many shared reads and writes), a *long but non-repetitive* one (having large read/write set), or even a *sparse* one (having many instructions, either shared accesses or not, inside the TX).

Section III will show in detail how these characteristics are helpful in understanding the performance of a TM system.

B. EigenBench

Pseudocode for the main functions in EigenBench is displayed in Figure 1. At its core, the benchmark is fairly simple; each thread performs a pre-defined number of transactions then exits. A transaction consists of a set number of transactional reads and writes (Lines 12 - 18), with some non-transactional "local" operations interspersed (Line 20). Additional local operations are performed between each transaction (Line 24). Parameters used in the code, most of which are arguments to the `test_core` function, are described in Table II. Note that throughout the code we occasionally modify the dummy variable `val` simply to prevent the compiler from optimizing code away.

Three separate arrays are accessed in the code. `Array1` is the *hot* array, meaning it is shared between all threads. `Array2` is the *mild* array, which is also accessed transactionally. Each thread accesses its own partition of `Array2`, however, so accesses will not cause conflicts. `Array3`, the *cold* array, is partitioned like `Array2` but is used for non-transactional accesses. As we discuss in the next section, using three distinct arrays allows us to control the contention between the transactions and the amount of influence the non-transactional code has on the caching behavior of the workload.

The `rand_actions` function decides if the transaction should perform a read or a write and whether to access the hot or mild array. This function guarantees that the precise number of reads and writes to each array, as given parameter values, are eventually performed but the order in which they are

TABLE I
ORTHOGONAL TM CHARACTERISTICS

Characteristic	Definition	Descriptive Adjectives
Concurrency	Number of concurrently running threads	high-concurrent/low-concurrent
Working-set size	Size of frequently used memory	wide/narrow
Transaction length	Number of shared accesses per TX*	long/short
Pollution	Fraction of shared writes to shared accesses	dirty/clean
Temporal locality	Probability of repeated address per shared access	repeating/non-repeating
Contention	Probability of conflict of a transaction	contentious/low-conflicting
Predominance	Fraction of shared access cycles to total execution cycles	significant/insignificant
Density‡	Fraction of non-shared cycles executed outside transactions to total non-shared cycles†	dense/sparse

* *Shared accesses* means reads/writes that should be protected by TM. *Shared cycles* is execution cycles consumed by such accesses.
† *Non-shared cycles* is execution cycles consumed by any other instruction than shared accesses.
‡ We define density as 1.0 if there is no non-shared instructions in the application, which does not happen in practice.

```
1  void test_core(tid, loops, pesist, lct, R1, W1, R2, W2
2      R3_i, W3_i, Nop_i, k_i, R3_o, W3_o, Nop_o, k_o) {
3    long val=0;
4    long total = W1 + W2 + R1 + R2;
5    for (i=0; i<loops; i++) {
6      Save_Random_Seed;
7      BEGIN_TM();
8      if (persist) Restore_Random_Seed;
9      (r1,r2,w1,w2) = (R1,R2,W1,W2);
10     Reset_History_Buffers;
11
12     for (j=0; j<total ; j++) {
13       (action, array) = rand_action(r1, w2, r2, w2);
14       index = rand_index(tid, lct, array);
15       if (action == READ)
16           val += TM_READ(array[index]);
17       else
18           TM_WRITE(array[index], val);
19       if ((j%k_i)==0)
20          val += local_ops(R3_i, W3_i, Nop_i, val, tid);
21     }
22     END_TM();
23     if ((i%k_o)==0)
24       val += local_ops(R3_o, W3_o, Nop_o, val, tid);
25 } }
26 static long A1, A2, A3, N;
27 static long *Array1, *Array2, *Array3;
28 void init_arrays() {
29   Array1 = malloc(A1 * sizeof(long));
30   Array2 = malloc(A2 * N * sizeof(long));
31   Array3 = malloc(A3 * N * sizeof(long));
32 }
33 (Action, Array) rand_action(r1, w1, r2, w2) {
34  // With uniform random probability based on r1,w1,r2,w2
35  // randomly choose one among: {(Read Array1),
36  // (Write Array1), (Read Array2), {Write Array2}}
37  // And decrease corresponding variable by one.
38 }
39 long rand_index(tid, lct, array) {
40  // With probability of lct, choose a saved index
41  // from the history buffer of array, or
42  // randomly choose an index from range
43  // (0~A1) or (tid*A2 ~ (tid+1)*A2)
44  // and save it to the history buffer of array
45 }
46 long local_ops(R3, W3, NOP, val, tid) {
47  // Perform R3 reads and W3 writes
48  // on Array3[tid*A3 ~ (tid+1)*A3] in random order.
49  // Then perform NOP number of nops.
50 }
```

Fig. 1. PseudoCode description of EigenBench.

performed is randomized. `local_ops` performs a given number of read/write operations on the cold array, then performs *nop*s. The call to this function on line 20 splits up the transactional accesses within a transaction, and the call on line 24 splits up the transactions themselves. Note that `k_i` and `k_o` are scalers; local operations can be either more (`k=1`) or less (`k>1`) numerous than shared operations.

The address accessed is expressed as an index to the selected array, which is determined by the function `rand_index`. Depending on the `lct` parameter which determines temporal locality, this will be either a random index in the range or an index previously used and saved in the history buffer.

Note that we abstracted out all instructions other than transactional reads and writes but captured their effects (e.g. instruction mix, ILP, or branches) simply with the parameters α and *nop*s. We did this because our goal was to qualify the performance characteristics of TM systems specifically, not those of general systems. We thus provide the key knobs that directly affect the TM systems. We remind the reader that to a TM system, a user application is fundamentally just a series of random reads and writes in concurrent transactions; EigenBench abstracts the application as such.

C. Deriving Eigen-characteristics from EigenBench

In this section, we explain how EigenBench can be used to generate a specific application execution pattern with the desired transactional characteristics.

Table III summarizes how the eigen-characteristics can be derived from EigenBench parameters[1], with the exception of *contention*. For *contention*, we use an approximate expected value, which we compute as follows. Let us denote the number of unique addresses in hot array accesses as W_1' and R_1'. [2] We start with the estimate that a given access will cause a conflict with probability $(N-1)*(W_1'))/A_1$, or the number of writes performed by all other transactions combined divided by the size of the shared array. [3] The complement of that event is an access not causing a conflict, which must happen $W_1'+R_1'$ times, giving us the probability of no conflicts. Finally, we take

[1]We defined working-set size based on per-thread memory usage. An alternative is to use total aggregated memory size. In that case, working-set size is derived as $A_1 + A_2 * N + A_3 * N$.

[2]W_1' is defined as follows: If $lct = 1$ then W_1' is 1. Else W_1' is $\lceil W_1 * (1 - lct) \rceil$. R_1' is defined similarly.

[3]This expression approximates the probability and is only valid when $A_1 >> W_1'$. For a better approximation, you can use $\hat{W}'$, the expected number of addresses occupied by N-1 other threads. This value can be obtained by the solution of the coupon collector's problem [16].

TABLE II
PARAMETERS USED IN EIGENBENCH

Name	Meaning	Name	Meaning	Name	Meaning
N	Number of Threads	R_1	Reads/tx of *Hot* array	W_3o	Writes of *Cold* array btwn TXs
S	Random Seed	W_1	Writes/tx of *Hot* array	Nop_i	No-ops between TM accesses
tid	Thread id	R_2	Reads/tx of *Mild* array	Nop_o	No-ops outside TX
loops	Number of TX per thread	W_2	Writes/tx of *Mild* array	K_i	Scaler for in-TX local ops
A_1	Size of Array1 (*Hot* array)	R_3i	Reads of *Cold* array inside TX	K_o	Scaler for out-TX local ops
A_2	Size of Array2 (*Mild* array)	W_3i	Writes of *Cold* array inside TX	persist	Restore random seed if violated
A_3	Size of Array3 (*Cold* array)	R_3o	Reads of *Cold* array btwn TXs	lct	Probability of address repetition

TABLE III
EIGENBENCH PARAMETERS USED TO DERIVE SPECIFIC CHARACTERISTICS

Characteristic	**Eigenbench Parameters**	**Characteristic**	**Eigenbench Parameters**
Concurrency	N	Working-set size	$A_1 + A_2 + A3$
Transaction length	$R_1 + R_2 + W_1 + W_2$ =(T_{len})	Pollution	$(W_1 + W_2)/T_{len}$
Temporal locality	lct	Contention	*see Equation (1)*
Predominance	$T_{len} * \alpha/(T_{len} * \alpha + C_{in} + C_{out})$	Density	$C_{out}/(C_{in} + C_{out})$
Read set Size*	$(R_1 + R_2) * (1 - lct)$	Write set Size*	$(W_1 + W_2) * (1 - lct)$

$N_{in} = ((R_{3i} + W_{3i}) * \alpha + Nop_i) * T_{len}/K_i, \quad O_{in} = \beta * T_{len} * (1 + (R_{3i} + W_{3i})/K_i), \quad C_{in} = N_{in} + O_{in}$
$N_{out} = ((R_{3o} + W_{3o}) * \alpha + Nop_o)/K_o, \quad O_{out} = \beta * (R_{3o} + W_{3o})/K_o, \quad C_{out} = N_{out} + O_{out}$
α: the average memory access latency β: overhead of random address generation and action decision in CPU cycles†

* We also include derivations for two important non-orthogonal characteristics: read set size and write set size.
† For simplicity of explanation, we assume (α, β) = (1,0) in the remaining of the paper.

the complement of that to get the probability of a conflict as shown in Equation 1. We remind the reader that the equation below estimates the degree of conflict induced by EigenBench with the given parameter values and not by any other general application. The validity of this equation will be demonstrated in Section III.

$$P_{conf} = 1 - \left(1 - \min\left\{1, \frac{(N-1)W_1'(1-lct)}{A_1}\right\}\right)^{W_1'+R_1'} \quad (1)$$

With EigenBench, it is always possible to adjust only one eigen-characteristic value while keeping the others fixed since there is always at least one free parameter for each attribute. For example, one can increase *transaction length* or *pollution* without increasing *contention* by controlling R_2 and W_2 only.

EigenBench is not limited to deploy uniformly characterized transactions across all threads. Rather, one can mix and match various characteristics in a single execution. For example, one thread can execute `test_core` function with one set of parameter values (e.g. long and dirty) while other threads execute with a different set of values (e.g. short and clean). Each thread can also execute different parameter sets over time, depending on modeled program 'phases'. Furthermore, one can slightly extend the code in Figure 1 such that the active parameters (e.g. R_1, W_1) become random variables themselves.

Finally, EigenBench can be easily extended to address other TM aspects such as nesting, strong atomicity and object-based TM. To address (the performance aspects of enforcing) strong atomicity, one can add accesses of shared arrays (Array1 and Array2) outside the transactions. A nested transaction can be implemented as recursive callings of test_core(). Object-based TM can also be addressed by replacing the long-typed arrays with an array of objects.

III. ORTHOGONAL ANALYSIS CASE STUDY: TL2 AND SWISSTM

In this section, we demonstrate how to perform orthogonal analysis using EigenBench to thoroughly analyze a TM system. Our evaluation features two high-performing STMs: TL2-x86 [5] (version 0.9.6 provided with STAMP [17] using the default GV4 versioned locks) and SwissTM [12] (version 2009-09-10 [18]). All systems are compiled with the `-m64 -O3` options. Our experiments were run on an HP ProLiant DL140 with two quad-core 2.33GHz Intel Xeon E5345 processors and 32GB of RAM. The E5345 has a total of 8MB of shared L2 cache. Throughout this section, we will italicize the names of eigen-characteristics as found in Table I.

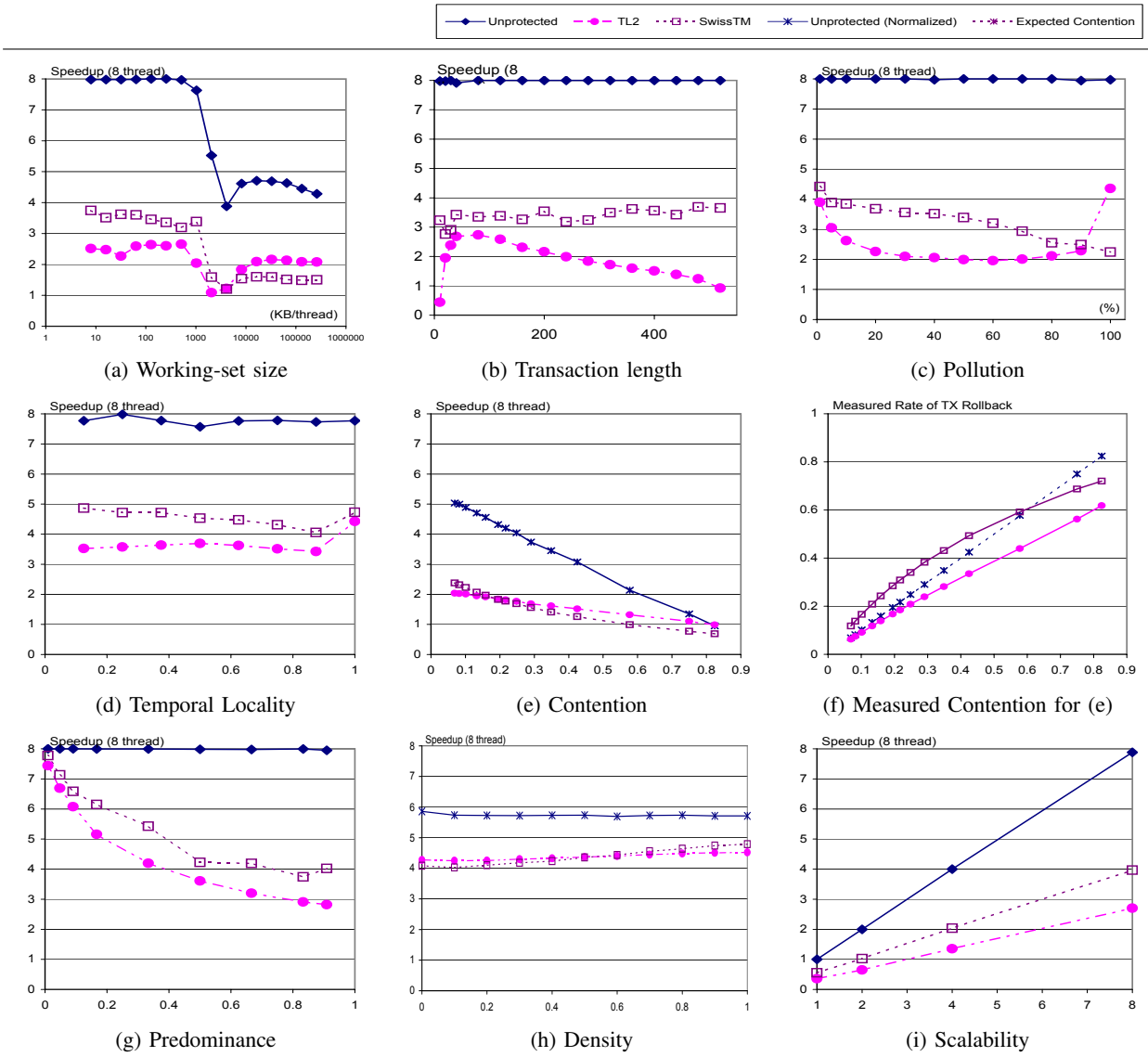
Since our analysis involves a high-dimensional search space, we only vary one dimension (i.e. characteristic) at a time while fixing the others to their *typical* values. Table IV displays our choices for typical values. We are interested in medium length (*transaction length* of 100) and relatively clean (10% *pollution*) transactions. We also use independent transactions (zero *contention*) when inspecting the overhead of the TM system; we will analyze the effect of *contention* separately. Finally, we fix *density* and *predominance* to be their maximum values (1.0) in order to focus on TM overhead only. Of course, one may select other typical values and perform similar analyses according to his interests (e.g., focusing on very short transactions with *transaction length* < 10). We remind the readers EigenBench allows for variation of any characteristic

TABLE IV
DEFAULT CHARACTERISTICS IN FIGURE 2

Characteristics	Value	Characteristics	Value	Characteristics	Value	Characteristics	Value
Concurrency	8	Working-set size	256 (KB/thread)	Transaction length	100	Pollution	0.10
Temporal locality	0.0	Contention	0.00	Predominance	1.00	Density	1.00

TABLE V
VARIED EIGENBENCH PARAMETERS IN FIGURE 2.

Graph	N	A1	A2	A3	R1	W1	R2	W2	R3i	W3i	R3o	W3o	lct	Range
(default)	8	0	32k	0	0	0	90	10	0	0	0	0	0	
(a)	-	-	Var	-	-	-	-	-	-	-	-	-	-	A_2=1k ... 32m
(b)	-	-	-	-	-	-	Var	Var	-	-	-	-	-	$W_2 + R_2$=10 ... 520
(c)	-	-	-	-	-	-	Var	Var	-	-	-	-	-	W_2=0 ... 100, $W_2 + R_2$=100
(d)	-	-	-	-	-	-	-	-	-	-	-	-	Var	lct=0.125 ... 1.0
(e),(f)	-	Var	Var	-	45	5	45	5	-	-	-	-	-	$A_1 + A_2$=32k, A_1=1k ... 24k
(g)	-	-	16k	16k	-	-	-	-	-	-	Var	-	-	R_{3o}=1 ... 100
(h)	-	512	31k	512	8	2	82	8	Var	-	Var	-	-	$N_{in} + N_{out}$=100, R_{3o}= 1 ... 90
(i)	Var	-	-	-	-	-	-	-	-	-	-	-	-	N=1... 8



(a) Working-set size (b) Transaction length (c) Pollution

(d) Temporal Locality (e) Contention (f) Measured Contention for (e)

(g) Predominance (h) Density (i) Scalability

Fig. 2. Orthogonal analysis of two TM systems using EigenBench. For (h), (0.24, 0.83) were used for Contention and Predominance, respectively.

while keeping others fixed since there is always one or more free parameters to accomplish this.

Figure 2 depicts the results of the orthogonal analysis. The x-axis denotes the value of the characteristic in question; Table V describes the values used in each graph. The y-axis denotes the speedup results with eight threads (higher is better). We plot three lines: TL2, SwissTM, and *Unprotected* speedup which represents the case of a multi-threaded execution without the protection of a TM system or even locks (i.e. TM reads/writes are mapped to direct loads/stores and TM begin/end to null statements). The unprotected execution thus serves as an upper bound of available performance, even though the bound is loose since no atomic execution is guaranteed. Note that EigenBench trivially allows the unprotected execution since it never suffers crashes or infinite loops due to a breach of atomicity; the same is not true for some benchmarks due to their dependence on complex data structures (e.g. rb-tree insertion). Also, note that when *contention* is zero, the unprotected execution then represents a hard upper bound.

Graph (a) shows the effect of different *working-set sizes*. The dramatic drop for all systems at 2MB/thread is an effect of the cache hierarchy; four threads execute concurrently on a quad-core chip with a shared 8MB L2 size of 8MB. This is seen even in the unprotected version because the performance is bounded by off-chip memory access. However, we also observe other interesting performance patterns for TL2 and SwissTM: When the working set fits on-chip, SwissTM performs much better than TL2 but performs worse when the working set outgrows the caches. This phenomenon seems to be caused by SwissTM's larger locktable size.

Graph (b) explores the effect of *transaction length*. The first four x-values shown are 10, 20, 30, and 40 with all subsequent values spaced evenly by 40. We see immediately that TL2 is extremely susceptible to transaction length; it performs poorly for very small transactions (10) and also degrades quickly again after a certain threshold value ($\tilde{1}20$). On the other hand, SwissTM handles all lengths equally well. The original SwissTM paper [12] explains how it handles different transaction lengths with different strategies.

Graph (c) shows the effect of *pollution*, or fraction of transactional writes. As we increase the number of writes, TL2's performance first drops quickly and then flattens; SwissTM's drops much more slowly and thus performs better overall due to its better write buffer structure.

Graph (d) shows the effect of *temporal locality*, or fraction of re-used addresses in a transaction. The graph shows that SwissTM's performance drops somewhat as we increase locality, indicating that the system is optimized for unique addresses. Both systems perform well when only a single address is used in a transaction (lct=1.0).

Previous discussion has focused on TM system overhead in the absence of *contention*. We now examine behavior when this is not the case. Before discussing the results, we note two things. First, the horizontal axis is the value of *expected contention*, as calculated by Equation 1. Second, since the unprotected execution does not stall or rollback and thus results in an overly loose bound, we normalize by dividing the measured execution time by one minus the expected amount of contention. This normalized execution time is still not a true bound, but a more reasonable approximation.

Graph (e) shows the effect of *contention*. One may notice that all speedups, including unprotected, are now far below 8, even when expected contention is very low. This is an effect of the system's ccNUMA architecture. Since the *hot* array (size of A_1 « 8MB) is shared by eight cores on two CPUs, about half of the accesses to that array will be served from the other CPU, even when the access may not result in a conflict for the current transaction.

The difference in measured speedup between TL2 and SwissTM in graph (e) is also interesting. We see that SwissTM performs only slightly better than TL2 when there are few conflicts and becomes worse with increasing degrees of contention. This is the result of two factors. First, SwissTM is more sensitive to the off-chip accesses induced by the ccNUMA architecture, as we saw in graph (a). Second, SwissTM rolls back more transactions than TL2, as will be seen in graph (f).

Graph (f) depicts the rate of transaction rollbacks as we vary the degree of expected contention. The graph shows that SwissTM has a higher rollback rate than both TL2 and the expected number of conflicts for the regular access patterns of the benchmark. This is because SwissTM detects conflict with quadword granularity while TL2 does so with word granularity and thus SwissTM exhibits false positives. The case of more irregular accesses will be discussed in Section V.

The results in graph (e) and (f) shows that SwissTM's performance is comparable to or better than TL2, even when it has twice high rollback rates. This suggests to TM designers that it makes sense to trade-off more false positives in conflict detection for less barrier overhead, since TM mostly targets low-conflicting applications.

Note that graph (f) serves to validate Equation (1). The graph clearly shows the high correlation between estimated degree of conflict (dotted line) and two actual violation rates reported by two STMs (solid lines); A TM design may allow a higher violation rate (e.g. false-positive filters) or a lower rate (e.g. transactional locks) than the 'real' conflicts.

Graph (g) shows the effect of *predominance*. As we decrease the *predominance* factor, the performance of both systems approaches that of the unprotected execution. Thus if shared accesses are rare, it makes no difference which system is used. As *predominance* increases, however, the plot indicates a measure of overall TM system overhead. SwissTM introduces less overhead than TL2 for the accompanying set of typical values.

Graph (h) depicts the effect of *density*. Note that we changed two of the typical values for this measurement. First, we set *predominance* to be 0.5. That is, in the entire program, there are as many non-TM instructions as there are TM instructions. In changing *density*, we change the proportion of non-TM instructions located outside transactions, with 1

being the maximum. Second, we choose a non-zero value for expected *contention*, 0.19, because it increases the re-execution penalty for adding more instructions inside transactions.

In graph (h), as we decrease *density* (i.e. putting more instructions into transactions), the performance drops slowly due to increased rollback penalty, but not much. The re-execution is cheaper than the first execution since much of fetched data still remain in its cache. Thus, a TM programmer may consider merging two short transactions interleaved by a short non-transactional section of code, since very small transactions may pose larger overhead in some TM systems (See graph (b)).

Graph (i) exhibits the scalability of two TM systems. Both systems scaled well with the non-conflicting transactions while SwissTM still maintained much less overhead as in previous graphs.

We now summarize our findings from the orthogonal analysis case study. We showed that TL2's performance drops quickly with long or dirty transactions (graphs (b) and (c)) while SwissTM is neutral to these characteristics. We also showed that SwissTM performs much better than TL2 only when the working-set size fits in the cache, but can get worse otherwise (graph (a)). Also SwissTM is shown to rollback more transactions than TL2 for regular data access pattern, mostly due to false positives in conflict detection. The results also indicate that it is reasonable to trade off barrier overhead and conflict detection accuracy (graphs (e) and (f)). Note that benchmark studies in the original SwissTM paper [12] did not reveal all these aspects but merely stated SwissTM performed better than TL2 in all cases.

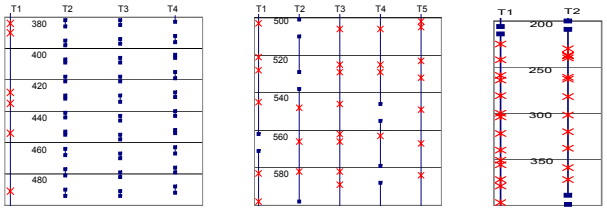
We omit further in-depth analysis on why one STM performs better than the other in specific cases, since this requires a thorough understanding of the details of the STM implementations and is out of scope of this paper. A TM system designer, however, would be able to use this analysis to properly evaluate trade-offs and gain more insight. One may even continue to analyze additional design spaces using different sets of typical values, or a mixture of transactions with different characteristics. Examples of such further exploration can be found in the Appendix.

IV. Pathology Generation

In the previous section, we showed how EigenBench can be used to analyze TM systems using orthogonal analysis. EigenBench can also be used to generate particular workloads which exercise TM systems in specific ways. In this section, we generate workloads that generate pathological TM behavior.

Transactional memory performance pathologies degrade performance by unnecessarily hindering the progress of transactions. Bobba et al. [2] showed that hardware TM performance notably improves by addressing the potential for pathologies in the system. Refer to the original paper [2] for detailed descriptions of all the pathologies.

Based on Bobba et. al.'s description of the pathologies, we built a set of EigenBench parameters that can generate



(a) StarvingElder (b) RestartConvoy (c) FriendlyFire

Fig. 3. Diagram of Observed Pathological Executions: The y-axis shows timestamps. Solid lines indicate the duration of a transaction; the dots represent begin/end for a transaction; and crosses are the re-start of a transaction.

each pathological behavior, depending on the underlying TM system implementation. The set of parameters is found in Table VI. Note that our choice of parameter values were arbitrary as long as they matched the description of the pathology situation in the paper [2]; any similar values produced the same pathology.

We present verification of the EigenBench parameters for three of the pathologies: StarvingElder, RestartConvoy and FriendlyFire. Our verification platform is the default TL2-x86, which uses lazy conflict detection, and its eager variant. Both versions use lazy version management and are provided with the STAMP benchmark suite, as in the previous section. For verification, we first add a simple global time stamp to the system; a dedicated thread increases this global variable on a regular basis. When a thread begins or ends a transaction, the event is stored in a local event history buffer with the current time stamp value. We analyze the data off-line to create a graphical representation of the execution sequence, as shown in Figure 3.

The pathology signatures are evident in the figure. In (a) StarvingElder, a long transaction (T1) cannot make progress due to shorter transactions (T2 $\sim$ T4) constantly violating it. In (b) RestartConvoy, a commit (e.g. T4 after 540) rolls back many other transactions and they all restart at about the same time. In (c) FriendlyFire, two threads (T1 and T2) violate each other continuously.

We omit verifications for other pathologies; SerializedCommit applies only to HTMs and all the others only to TM systems that use eager conflict detection and eager version management. If provided these systems, however, one could easily utilize EigenBench with its simplicity and flexibility to generate the remaining pathologies.

V. Characterization of Actual Workloads

In previous sections, we have shown that by varying the eigen-characteristics using EigenBench, we can effectively analyze a TM system. We will now demonstrate that there is an actual mapping from a real application to a set of characteristics. Of course, real applications usually have a mix of various transaction types and exhibit complex memory access patterns. In this section, we address these while answering the following three questions:

TABLE VI
EIGENBENCH PARAMETERS FOR GENERATION OF PATHOLOGICAL CASES

Pathology	**EigenBench Parameters**													
(TM Design Point)	N	A1	A2	A3	R1	W1	R2	W2	R3i	W3i	R3o	W3o	lct	persist*
FriendlyFire (EL)	2	1k	16k	8k	100	20	400	0	200	0	0	0	0	1
StarvingElder (LL)	1	1k	-	8k	128	32	0	0	0	0	100	100	0	0
	7	1k	-	8k	2	2	0	0	0	0	100	100	0	0
RestartConvoy (LL)	8	8	4k	8k	4	2	20	20	5	0	100	100	0	1
SerializedCommit(LL)	8	-	16k	8k	0	0	10	500	0	0	0	0	0	0
StarvingWriter (EE)	1	32	-	1m	0	30	0	0	500	0	0	0	0	0
	7	32	-	-	30	0	0	0	0	0	0	0	0	0
FutileStall (EE)	8	1k	16k	-	80	20	10	10	0	0	0	0	0	0
DuelingUpgrade (EE)	2	128	-	2m	80	10	0	0	20	0	0	0	0.75	0

* The *persist* option compels the transaction to repeat the same address sequence if it retries (see Fig. 1). This causes some pathologies occur more frequently.

- Can we get (or approximate) eigen-characteristics for real applications?
- If so, what characteristic (or sets of characteristics) dictates the performance of those applications?
- Can Eigenbench reproduce the performance behavior of those applications, given their eigen-characteristics?

In this analysis, we use applications from the STAMP benchmark suite [3]. We are first able to identify some of the eigen-characteristics through profiling: *transaction length, working-set size, pollution, and locality*.When measuring *locality*, we use a unit of four words. STMs based on manual instrumentation provide an easy way to achieve this profiling: We simply log shared accesses in single-threaded execution and perform off-line analysis. Appropriate values for *predominance* and *density* are estimated from application documentation or using source code analysis, although exact values could be obtained via instrumentation-based profiling or simulation. Recall from Section III that *predominance* is simply a scaling factor for all TM systems, and performance sensitivity to *density* is small. Finally, since an exact value for *contention* is hard to capture, we use the rollback percentage, reported by the TM system, as a good approximation. Table VII summarizes the results.

Please note that certain applications exhibit large variations in characteristic values; some could be described as having several modes of operation. For example, the first column in Figure 4 shows a histogram of transaction lengths for the five STAMP applications featured. The transaction lengths of Genome and Intruder have long "tails" in the graph, while Labyrinth is rather uniformly distributed. On the other hand, Vacation-Low has a normal distribution and SSCA2 is single-valued. In the second column of Figure 4, we provide diagrams of the access frequency per address region; this illustrates the memory access pattern and working-set size of each application.

Our next question is determining what characteristics dictate the performance behaviors of these applications. We answer this question while reproducing these behaviors using EigenBench. In this study, we manually obtained EigenBench parameters that capture the collected eigen-characteristics in Table VIII, since they are discrete abstractions of the continuous characteristics shown in the first two columns of Figure 4. Note that a single parameter set was enough to abstract some applications, while multiple sets were needed for others, depending on the variance of the characteristics.

We present the EigenBench result in the third column of Figure 4; the graphs display the speedup and transaction rollback rates for each application. We plot four lines for each graph: two solid lines from the original application executed with TL2 and SwissTM as well as two dotted lines from the EigenBench executions with the parameters in Table VIII.

For **Genome**, the governing characteristic was *transaction length* or, more precisely, a mixture of different lengths. In Genome, the dominant transaction length is relatively small ($<$ 100); however, it has a long tail, meaning larger transaction lengths are used with nontrivial frequency. We capture this dynamic property by using three different parameter sets (Table VII); this was essential to capture the performance behavior. SwissTM's better scalability with 8 threads is a result of its sophisticated conflict resolution mechanism which prevents overly frequent aborting of long transactions. Also note that, due to this mechanism, SwissTM now exhibits a lower rollback rate than TL2, unlike the analysis in Section III.

Vacation-Low exhibits very different key characteristics. *Transaction length* is somewhat normally distributed, but the application uses a much wider range of memory. As a result, the limitation becomes cache miss latency resulting from large *working-set sizes*. We captured this with a single set of parameter values as shown in Table VII.

Labyrinth has a uniform distribution over *transaction length* and a narrow memory footprint, as in the case of Genome. Although the transaction size is relatively long for this application, it has an extremely low *density* (meaning many non-TM instructions inside transactions) and low *predominance* (many non-TM operations overall). This suggests that it is not the overhead of transactional read/write operations, but rather the efficiency of conflict detection that governs its performance.

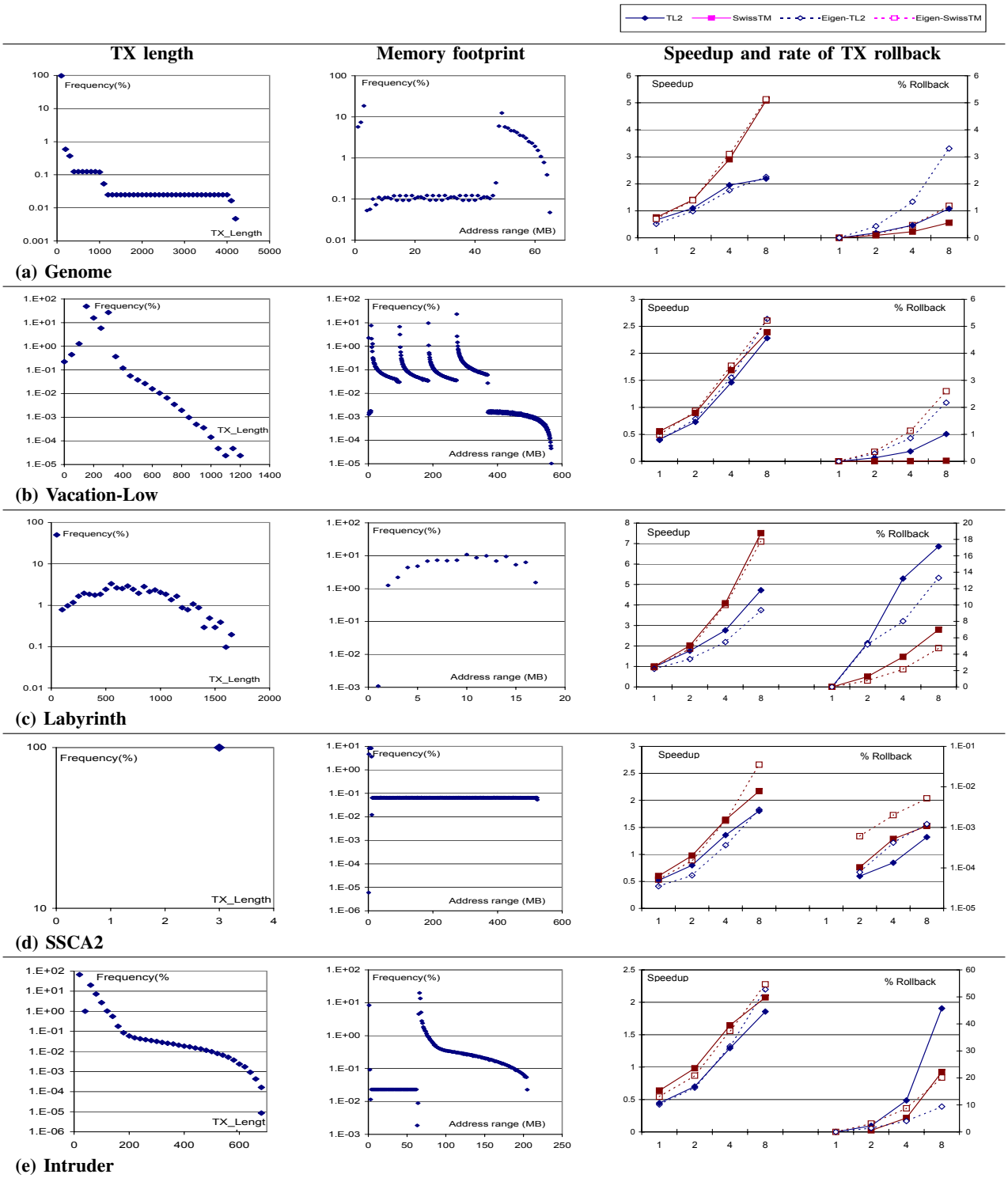


Fig. 4. Characteristics and Performance of STAMP Applications and EigenBench: The first two columns are histograms; each presents an Eigen-characteristic (TX length and Working-set size) for the application in each row. The last column compares the measured performance and rollback ratio of the original STAMP application (solid lines) and EigenBench executions with parameters set as in Table VIII (dotted lines). Note that the Eigen-characteristic values for EigenBench execution can be deterministically calculated from those parameters.

TABLE VII
EIGEN-CHARACTERISTICS OF STAMP APPLICATIONS

Application†	**Working-set**	**TX Length**	**Pollution**	**Locality**	**Contention**	**Predominance**	**Density***
Genome	20 MB	88, (1..4000)	5%	0.58	0.5%	High	High
Vacation-low	256 MB	226, (1.. 1239)	2%	0.59	0.2%	High	High
Vacation-high	256 MB	180, (1.. 1878)	4%	0.55	0.4%	High	High
Labyrinth	16 MB	357, (3..1688)	50%	0.77	5%	Low	Low
SSCA2	400 MB	{ 3 }	33%	0.33	0.0005%	Low	High
Intruder	20 MB	24, (3..680)	5%	0.52	22%	Low	High
Kmeans-low	8MB (8KB)	{2, 66}	50%	0.81	25%	Low	High
Kmeans-high	8MB (8KB)	{2, 66}	50%	0.81	43%	Low	High
Yada	200MB	45 (1..2194)	28%	0.47	22%	High	High

* For working-set of Kmeans, the number inside parenthesis is the amount of memory addresses accessed inside TX. For TX length, we present average value and range. For pollution and locality, we present measured average if the distribution is unimodal, or top n-mode values if multi-modal. Contention is estimated by the minimum value of the percentage of transactions that roll back on the given TM systems with 8 threads. We loosely estimated predominance and density from source code analysis. Also see the first two columns of Figure 4 which display variance of tx-length and memory footprint.

† Bayes failed to execute in our 64-bit environment and is excluded from this study.

TABLE VIII
EIGENBENCH PARAMETERS FOR MIMICKING SELECTED STAMP APPLICATIONS.

App.	A1	A2*N	A3	mix(%)	R1	W1	R2	W2	R3o	W3o	R3i	W3i	LCT
Genome	256k	2m	256k	97%	28	2	65	5	20	20	0	0	0.5
				2%	145	5	340	10	80	80	0	0	0.5
				1%	770	30	1660	40	500	500	0	0	0.5
Vacation-low	512k	32k	64k	100%	198	2	47	3	0	0	0	0	0.6
Labyrinth	128k	2m	64k	55%	10	5	0	0	5	5	0	0	0.7
				15%	25	25	50	50	0	0	600	600	0.7
				15%	25	25	275	275	0	0	600	600	0.7
				15%	25	25	650	650	0	0	600	600	0.7
SSCA2	32m	8k	64k	100%	1	1	1	0	15	0	0	0	0
Intruder	1k	1k	64k	70%	8	2	0	0	2	2	0	0	0.5
				28%	18	2	27	3	4	4	0	0	0.5
				2%	20	10	90	10	10	10	0	0	0.5

SSCA2 shows a very uniform data access pattern as illustrated in the second column of Figure 4 and Table VII. In this case, short *transaction lengths* and large *working-set size* govern the performance.

Intruder displays a variety of *transaction lengths* and a wide range of addresses accessed. It also displays a large amount of *contention*, as shown in Table VII. The performance is mainly limited by the contention and short transactions.

Overall, EigenBench was able to closely approximate TM application behavior by reproducing key eigen-characteristics of the applications. Please note that all of these applications exhibit complex (pointer-chasing) memory access patterns, but the approximations were still reasonably close. We remind the reader that the goal of our analysis is not to provide 100% identical replay of the target application; we are interested in capturing a few key characteristics and simulating those characteristics with EigenBench.

We omit graphs of other applications' behavior but discuss their key characteristics here. **Vacation-High** has a large *working-set size* similar to **Vacation-Low**, and this governs its behavior. However, it exhibits more *contention* than Vacation-Low. Both versions of Kmeans have very narrow *working-set sizes*, short transactions and high *locality*; these further emphasize the effect of barrier overhead. Like Genome, **Yada** shows large variance in *transaction length*, but it has a much wider memory footprint. Also, transactions are more contentious and more dirty than in Genome.

To summarize, we have shown that one can obtain (mixed sets of) eigen-characteristics for real applications, but there are only a few key characteristics that dictate the performance behavior of an application. Along with results from an orthogonal analysis (as performed in Section III), knowledge about the characteristics of TM applications can explain the performance behavior of such applications as exemplified in this section. A TM developer can use this information to further improve a system. For example, to improve the performance of the Vacation application, one must minimize the TM system's impact on cache pressure while efficiently handling long transactions.

We have also shown that using EigenBench, one can reasonably model certain application behaviors given (sets of) eigen-characteristics. This is clearly evident in the similarities between the EigenBench and TL2/SwissTM executions, displayed in the third column of Figure 4. Conversely, the performance behavior of EigenBench with a certain set of parameters accurately represents the performance behavior of complex applications which exhibit those characteristics. In other words, since the eigen-characteristics successfully explain performance behavior of a TM application, a through

exploration using EigenBench allows analysis of a TM system across an extremely wide range of applications not covered by other benchmarks.

VI. Conclusion

Previous transactional memory benchmark suites and microbenchmarks have often lacked the ability to isolate the key attributes of an application and determine how each affects the performance of the TM system. Additionally, there is little consensus in the literature on which application characteristics accurately and independently represent realistic application behavior. In this paper, we proposed eigen-characteristics, a set of orthogonal application characteristics which are very useful in analyzing a TM system. Together, the eigen-characteristics capture the dominant behavior of TM applications. To provide a simple way to explore these attributes, we presented EigenBench, a lightweight yet powerful parameterizable microbenchmark. As a demonstration, we used EigenBench to analyze two prominent STM systems. EigenBench allowed us to vary each characteristic independently, providing insight on how they affected the systems in the absence of other interfering factors. In addition, we showed that the eigen-characteristics can successfully specify real TM applications by using EigenBench to accurately reproduce the behavior of several STAMP applications. In doing so, we demonstrated a concrete mapping between real applications and (sets of) eigen-characteristics. We also showed how EigenBench can be used to generate important specific execution patterns by reproducing three TM pathological cases.

Acknowledgements

This work is supported by DOE contract, Sandia order 942017; Army contract AHPCRC W911NF-07-2-0027-1; DARPA contract, Oracle order 630003198; and Stanford PPL affiliates program, Pervasive Parallelism Lab.

References

[1] J. Larus and R. Rajwar, *Transactional Memory*. Morgan Claypool Synthesis Series, 2006.

[2] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *ISCA '07*.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08*, September 2008.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *ASPLOS'06*.

[5] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*. Springer, March 2006, pp. 194–208.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *ASPLOS '09*.

[7] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *PPoPP '06*.

[8] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg, "McRT–STM: A high performance software transactional memory system for a multi-core runtime," in *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[9] G. Kestor, S. Stipic, O. Unsal, A. Cristal, and M. Valero, "Rms-tm: A transactional memory benchmark for recognition, mining and synthesis applications," in *TRANSACT'09: 4th workshop on transactional computing*.

[10] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis, "Lee-tm: A non-trivial benchmark suite for transactional memory," *Lecture Notes in Computer Science*, vol. 5022, pp. 196–207, 2008.

[11] R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: A Benchmark for Software Transactional Memory," in *Proceedings of the Second European Systems Conference (EuroSys2007)*, 2007.

[12] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2009, pp. 155–165.

[13] C. Hughes, J. Poe, A. Qouneh, and T. Li, "On the (dis)similarity of transactional memory workloads," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 108–117.

[14] J. Poe, C. Hughes, and T. Li, "TransPlant: A Parameterized Methodology For Generating Transactional Memory Workloads," in *MASCOTS'09: IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*.

[15] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun, "Characterization of TCC on Chip-Multiprocessors," in *PACT '05*.

[16] I. Adler, S. Oren, and S. Ross, "The coupon-collector's problem revisited," *Journal of Applied Probability*, vol. 40, no. 2, pp. 513–518, 2003.

[17] "STAMP: Stanford transactional applications for multi-processing," http://stamp.stanford.edu.

[18] "SwissTM," http://lpd.epfl.ch/site/research/tmeval#swisstm.

[19] "EigenBench," http://ppl.stanford.edu/eigenbench.

Appendix

In this section, we provide a short introduction to using EigenBench for analysis of a TM system. Eigenbench is publicly available from our website [19]. We provide source code that can be easily executed in conjunction with any STM or HTM implementation or simulation. The package also includes specific EigenBench parameters discussed in the paper.

(1) Orthogonal analysis: The main usage of EigenBench should be the orthogonal overhead analysis as in Section III. First, one may specify a typical transaction description of interest (e.g. transaction length and pollution), making sure to use non-conflicting characteristics. At a minimum, one should explore working-set size, transaction length, pollution and scalability, and compare the results against *unprotected* execution (see Section III). This will reveal the true overhead induced by the TM system. Results for locality, predominance and density may be omitted in the report if they don't provide further insights beyond those reported in this paper. Finally, performance under contention and measured rollback rates should be explored.

(2) Mixed transactions: Optionally, one may want to analyze performance under non-uniform transactional characteristics (e.g. long transactions mixed with short transactions). For this purpose, we provide multiple sets of parameters in our distribution package, based on our application analysis. Given very mixed parameters, it is not easy to obtain an analytic model for the true degree of conflict, as shown in equation (1). For such mixed parameters, we suggest Monte Carlo estimation.

(3) Pathology: Optionally, one may also test TM system performance under pathological transactions generated by EigenBench parameters. This can verify if the TM system is immune to the pathologies or susceptible to them.

(4) Explanation of TM application behavior: We believe that the above analysis should provide enough information to explain a certain TM application's performance behavior, as long as one knows its eigen-characteristics. However, one should also check if the application is governed by non-TM aspects (e.g. Amdhal limit, thread sync operation outside TX, etc.), which may not be explained by any TM characteristics.

Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications

Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi, and Tao Li
Intelligent Design of Efficient Architecture Lab (IDEAL)
University of Florida, Gainesville, Florida, USA
{nil, sramkumar, mjoshi}@ufl.edu, taoli@ece.ufl.edu

***Abstract*— The GPUs are emerging as a general-purpose high-performance computing device. Growing GPGPU research has made numerous GPGPU workloads available. However, a systematic approach to characterize these benchmarks and analyze their implication on GPU microarchitecture design evaluation is still lacking. In this research, we propose a set of microarchitecture agnostic GPGPU workload characteristics to represent them in a microarchitecture independent space. Correlated dimensionality reduction process and clustering analysis are used to understand these workloads. In addition, we propose a set of evaluation metrics to accurately evaluate the GPGPU design space. With growing number of GPGPU workloads, this approach of analysis provides meaningful, accurate and thorough simulation for a proposed GPU architecture design choice. Architects also benefit by choosing a set of workloads to stress their intended functional block of the GPU microarchitecture. We present a diversity analysis of GPU benchmark suites such as *Nvidia CUDA SDK*, *Parboil* and *Rodinia*. Our results show that with a large number of diverse kernels, workloads such as S*imilarity Score*, *Parallel Reduction*, and *Scan of Large Arrays* show diverse characteristics in different workload spaces. We have also explored diversity in different workload subspaces (e.g. memory coalescing and branch divergence). *Similarity Score*, *Scan of Large Arrays*, *MUMmerGPU*, *Hybrid Sort,* and *Nearest Neighbor* workloads exhibit relatively large variation in branch divergence characteristics compared to others. Memory coalescing behavior is diverse in *Scan of Large Arrays*, *K-Means*, *Similarity Score* and *Parallel Reduction*.**

I. Introduction

With the increasing numbers of cores per CPU chip, the performance of microprocessors has increased tremendously over the past few years. However, data-level parallelism is still not well exploited by general-purpose chip multiprocessors for a given chip area and power budget. With hundreds of in-order cores per chip, GPU provides performance throughput on data parallel and computation intensive applications. Therefore, a heterogeneous microarchitecture, consisting of chip multiprocessors and GPUs seems to be good choice for data parallel algorithms. Nvidia CUDA™ [19], AMD stream™ [23] and OpenCL [49] programming abstractions have provided data parallel application development thrust by reducing significant amount of development effort.

Emerging CPU-GPU heterogeneous multi-core processing has motivated the computer architecture research community to study various microarchitectural designs, optimizations, and analysis for GPU, such as, an efficient GPU on-chip interconnect arbitration scheme [1], more efficient GPU SIMD branch execution mechanisms [2], technique to diverge on a memory miss to better tolerate memory latencies in SIMD cores [4] etc. However, it is largely unknown whether the currently available GPGPU workloads are capable of evaluating the whole design space. An ideal evaluation mechanism must be *accurate*, *thorough* and *realistic*. *Accuracy* is provided by the applicability of the deduced conclusions that are minimally affected by the chosen benchmarks. A *thorough* evaluation mechanism covers a large amount of diverse benchmarks, where each workload stresses different aspects of the design. *Realistic* evaluation guarantees that for lesser number of simulations *thoroughness* can be achieved. *Realistic* evaluation narrows down the workload simulation space (time as well), while keeping the simulation fidelity and actual conclusion within an acceptable threshold.

In order to achieve the above goals, we propose a set of GPU microarchitecture agnostic GPGPU workload characteristics to accurately capture workload behavior and use a wide range of metrics to evaluate the effectiveness of our characterization. We employ principal component analysis [7] and clustering analysis methods [8], which have been shown [9, 10, 11, 12] to be effective in analyzing benchmark suites such as SPEC CPU2000 [13], SPEC CPU2006 [51], MediaBench [14], MiBench [15], SPLASH-2 [16], STAMP [17] and PARSEC [18] benchmarks.

This paper makes the following contributions:

- We propose a set of GPGPU workload characterization metrics. Using 38×6 design points, we show that these metrics are independent of the underlying GPU microarchitecture. These metrics will allow GPGPU researchers to evaluate the performance of emerging GPU microarchitectures regardless of their microarchitectural improvements. Though, we characterize the GPGPU workloads using Nvidia GPU microarchitecture [19], the conclusions drawn here are mostly applicable to other GPU microarchitectures such as AMD ATI [23].
- Using the proposed GPGPU workload metrics, we study the similarities between existing GPGPU kernels and observe that they often stress the same bottlenecks. We show that removing redundancy can significantly save simulation time.
- We provide workload categorization based on various workload subspaces like, divergence characteristics, kernel characteristics, memory coalescing etc. We categorize different workload characteristics according to their importance. We also show that available workload space is most diverse in terms of branch divergence characteristics and least diverse in terms of thread-batch level coalescing behavior. Relative diversity among *Nvidia CUDA SDK* [28], *Parboil* [26] and *Rodinia* [25] benchmark suites is also explored.

The rest of this paper is organized as follow. Section II provides the background of GPU microarchitecture along with CUDA [19] programming model. Section III describes the proposed GPU kernel characterization metrics as well as the statistical methods used for data analysis. Section IV describes our experimental

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

methodologies including simulation framework, benchmarks and evaluation metrics. Section V presents our experimental results. Section VI highlights the related research. Section VII concludes the paper.

II. An Overview of GPU Microarchitecture and Programming Model

Fig.1. shows the general-purpose GPU microarchitecture where unified shader cores work as in-order multi-core streaming processor [19]. In each streaming multiprocessor (SM), there are several in-order cores, each of which is known as single processor (SP). The number of SPs per SM varies with the generations of GPU. Each SM consists of an instruction cache, a read-only constant cache, a shared memory, registers, a multi-threaded instruction fetch and issue unit (MT unit), several load/store and special functional units [21, 3]. Two or more SMs are grouped together to share a single texture processor unit and L1 texture cache among the SMs [21]. An on-chip interconnection network connects different SMs to L2 cache and memory controllers. Note that, in addition to CPU main memory, the GPU device has its own external memory (off-chip), which is connected to the on-chip memory controllers. The host CPU and GPU device communicate with each other through the PCI express bus. Some high-end GPUs also have a L2 cache, configurable shared memory, L1 data cache and error-checking-correction unit (ECC) for memory.

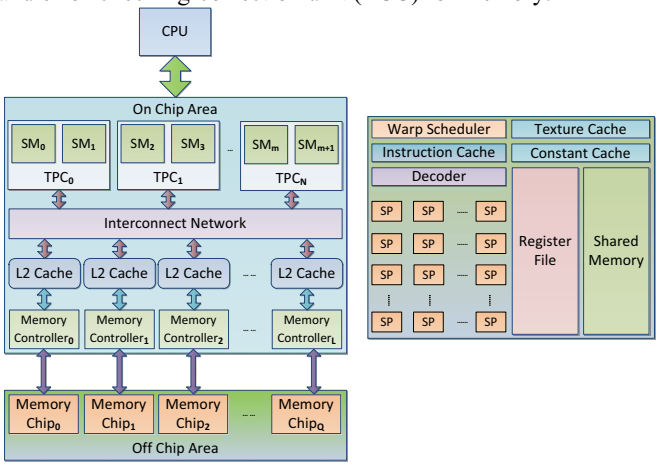


Fig. 1. GPU Microarchitecture (left), Streaming Multiprocessor (right)

General-purpose programming abstractions, such as Nvidia CUDA and AMD Stream, are often used to facilitate GPGPU application development. In this paper, we have used Nvidia CUDA programming model but some of the basic constructs will hold for most programming models. Using CUDA, thousands of parallel and concurrent lightweight threads can be launched by the host CPU and those are grouped together in an execution entity called *blocks*. Different *blocks* are grouped into an entity called a *grid*. During execution, a particular *block* is assigned to a given SM. *Blocks* running on different SMs cannot communicate with each other. Based on the available resources in a SM, one or more *blocks* can be assigned to the same SM. All the registers of the SM are allocated to different threads of different *blocks*. This leads to faster context switching among the executing threads.

Usually 32 threads from a *block* are grouped into a warp that executes the same instruction. However, this number can vary depending upon the GPU generation. Threads in GPU execute based on the single-instruction-multiple-thread (SIMT) [20] model. The execution of branch instructions may cause some threads to jump, while others fall through in a warp. This phenomenon is called warp divergence. Threads in a diverged warp execute in serial fashion. Diverged warps possess inactive thread lanes. This reduces the thread activity of the kernel below 100%. Kernels with less divergent warps work efficiently for both independent scalar threads as well as data parallel coordinated threads in SIMT mode. Load/store requests issued by different threads in a warp get coalesced in load/store unit into a single memory request according to their access pattern. Memory coalescing improves the performance by hiding the individual smaller memory accesses in a single large memory access. This phenomenon is termed as *intra-warp memory coalescing*. Threads in a *block* execute in synchronized fashion.

III. Understanding GPGPU Kernels: Characteristics and Methodology

A. GPGPU Kernel Characteristics

Emerging GPGPU benchmarks are not well understood; because these workloads are significantly different from conventional parallel applications that utilize control-dominated heavy weight threads. GPGPU benchmarks are inherently data parallel with light weight kernels.

Until now, the effects of different workload characterization parameters have not been explored to understand the fundamentals of GPGPU benchmarks. In this paper, we propose and characterize the GPGPU workload characteristics according to microarchitectural viewpoint. Table I lists the features that specify the GPGPU specific workload characteristics. Table II lists the generic workload behaviors. Index in Table I refers to the dimensions of the data in the input matrix (129×38).

To capture overall workload stress on the underlying microarchitecture, we have used *dynamic instruction count*, which is the total number of instructions executed on a GPU inclusive of all streaming multiprocessors. The *dynamic instruction count per kernel* provides per-kernel instruction count of the workload. *Average instruction count per thread* captures average stress applied to each streaming processors during kernel execution. To express the degree of parallelism we proposed *thread count per kernel* and *total thread count* metrics. Above mentioned metrics represent the *kernel stress subspace*.

Instruction mix is captured using *floating-point instruction count*, *integer instruction count*, *special instruction count,* and *memory instruction count* metrics. These parameters provide an insight of the usage of different functional blocks in the GPU. Inherent capability of hiding memory access latency is encapsulated within *arithmetic intensity*. Memory type based classification is captured using *memory instruction count*, *shared memory instruction count*, *texture memory instruction count*, *constant memory instruction count*, *local memory instruction count,* and *global memory instruction count*. *Local memory instruction count* and *global memory instruction count* encapsulate the information of off-chip DRAM access frequency that radically improves or degrades the performance of the workload. *Branch instruction count* provides the control flow behavior. Due to warp

divergence, the frequency of branch instructions plays a significant role in characterizing the benchmarks. *Barrier instruction count* and *atomic instruction count* show synchronization and mutual exclusion behavior of the workload, respectively. Branch and divergence behaviors are characterized by *percentage of divergent branches*, *number of divergent warps*, *percentage of divergent branches in serialized section,* and *length of the serialized section*. Branches due to loop constructs in a workload or thread-level separate execution paths do not contribute to warp divergence. Number of threads in the taken path or fall-through path defines the length of the serialized section. *Number of divergent warps* is defined as total number of divergent warps during the execution of the benchmark. It expresses the amount of SIMD lane inactivity in terms of idle SPs in SM. *Length of serialized section* is the number of instructions executed between a divergence point and corresponding convergence point.

TABLE I
GPGPU WORKLOAD CHARACTERISTICS

Index	Characteristics	Synopsys
1	Special instruction count (I_{sop})	Total number of special functional unit instructions executed.
2	Parameter memory instruction count (I_{Par})	Total number of parameter memory instructions.
3	Shared memory instruction count (I_{shd})	Total number of shared memory instructions.
4	Texture memory instruction count (I_{tex})	Total number of texture memory instructions.
5	Constant memory instruction count (I_{const})	Total number of constant memory instructions.
6	Local memory instruction count (I_{loc})	Total number of local memory instructions.
7	Global memory instruction count (I_{glo})	Total number of global memory instructions.
8 – 17	Percentage of divergent branches (D_{bra})	Ratio of divergent branches to total branches.
18 – 27	Number of divergent warps (D_{wrp})	Total number of divergent warps in a benchmark.
28 – 37	Percentage of divergent branches in serialized section (B_{ser})	Ratio of divergent branches to total branches inside a serialized section.
38 – 47	Length of serialized section (l_{serial})	Instructions executed between a divergence and a convergence point.
48 – 57	Thread count per kernel (T_{kernel})	Count of total threads spawned per kernel.
58	Total thread count (T_{total})	Count of total threads spawned in a workload.
59 – 63	Registers used per thread (Reg)	Total number of registers used per thread.
64 – 68	Shared memory used per thread (Sh_{mem})	Total amount of shared memory used per thread.
69 – 73	Constant memory used per thread (c_{mem})	Total amount of constant memory used per thread.
74 – 78	Local memory used per thread(l_{mem})	Total amount of local memory used per thread.
79	Bytes transferred from host to device (H2D)	Bytes transferred from host to device using *cudaMemcpy*() API.
80	Bytes transferred from device to host(D2H)	Bytes transferred from device to host using *cudaMemcpy*() API.
81	Kernel count (K_n)	Number of kernel calls in the workload.
82 – 91	Row locality (R)	Average number of accesses to the same row in DRAM.
92 – 101	Merge miss (M_m) [6]	Cache misses/uncached accesses can be merged with ongoing request.

TABLE II
GENERIC WORKLOAD CHARACTERISTICS

Index	Characteristics	Synopsys
102	Dynamic instruction count (I_{total})	Dynamic instruction count for all the kernels in a workload.
103 – 112	Dynamic instruction count per kernel (I_{kernel})	Per kernel split up of the dynamic instructions executed in a workload.
113 – 122	Average instruction count per thread (I_{avg})	Average number of dynamic instructions executed by a thread.
123	Floating point instruction count (I_{fop})	Total number of floating point instructions executed in a given workload.
124	Integer instruction count (I_{op})	Total number of integer instructions executed in a given workload.
125	Memory instruction count (I_{mop})	Total number of memory instructions.
126	Branch instruction count (I_b)	Total number of branch instructions.
127	Barrier instruction count (I_{bar})	Total number of barrier synchronization instructions.
128	Atomic instruction count(I_{ato})	Total number of atomic instructions.
129	Arithmetic Intensity (A_i)	Arithmetic and logical operations per memory operation across all kernels.

Thread level resource utilization information is retrieved using metrics such as *registers per thread*, *shared memory used per thread*, *constant memory used per thread* and *local memory used per thread*. For example, in a SM registers are allocated to a thread from a pool of registers; minimum granularity of thread allocation is a *block* of threads. Too many threads per *block* may exhaust the register pool. The same holds true for shared memory. Thread-level local arrays are allocated into off-chip memory arrays; designated by *local memory used per thread*. CPU-GPU communication information is gathered from *bytes transferred from host to device* and *bytes transferred from device to host*. The former provides a notion of off-chip memory usage in GPU. The scenario when the data in the GPU is computed by itself without using any input data from CPU is distinct from the scenario when GPU processes the data received from CPU and returns it back to CPU. *Kernel count* tells the count of different parallel instruction sequences present in the workload. These metrics provide *kernel characteristics subspace*.

In order to capture intra-warp and inter-warp memory access coalescing, we use *row locality* [1] and *merge miss* [6] as described in Table I. Merge miss reveals number of cache misses or uncached accesses that can be merged into another in-flight memory request [6]. These accesses can be coalesced into a single memory access and memory bandwidth can be improved. DRAM row access locality in the data access pattern is captured before it is shuffled by the memory controller. We assume GDDR3 memory in this study. With different generations of GDDR memory these numbers may slightly differ, but it still remains microarchitecture agnostic. These two metrics comprise the *coalescing characteristics subspace*.

Some of the previously mentioned parameters are correlated with each other. For example, if a workload exhibits large integer instruction count, it will also show large dynamic instruction count. If *percentage of divergent branches in a serialized section* is high then it is probable that the *length of serialized section* will also be high. This nature of high data correlation justifies the use of PCA analysis.

B. Statistical Methods

A brief description of *principal component analysis* and

hierarchical clustering analysis is provided here.

B.1 Principal Component Analysis

Principal component analysis (PCA) is a multivariate data analysis technique to remove the correlations among the different input dimensions and to significantly reduce the data dimensions. In PCA analysis, a matrix is formed using different columns representing different input dimensions and different rows representing different observations. PCA transforms these observations into principal component (PC) domain, where PCs are linear combination of input dimensions. Mathematically, a dataset of n correlated variables has k^{th} observation of the form, $D_k \equiv (d_{0k}, d_{1k}, d_{2k}, d_{3k}, ..., d_{(n-1)k})$. The k^{th} observation in PC domain is represented as $P_k \equiv (p_{0k}, p_{1k}, p_{2k}, p_{3k}, ..., p_{(n-1)k})$ where $p_{0k} = c_0 d_{0k} + c_1 d_{1k} + c_2 d_{2k} + c_3 d_{3k} + ... + c_{(n-1)} d_{(n-1)k}$. Terms like $c_0, c_1, ..., c_{n-1}$ are referred to as *factor loading* and those are selected in PCA process to maximize the variance for a particular PC. Variance of i^{th} PC (σ^2_i) has the property of $\sigma^2_{i-1} < \sigma^2_i < \sigma^2_{i+1}$. PCs are arranged in the decreasing order of eigen values. Eigen values describe the amount of information present in a principal component. *Kaiser criteria* [46] suggests considering all the PCs that have eigen value greater than 1. In general, certain numbers of first principal components are chosen to obtain 90% of the total variance as compared to original data variance. Usually the count is much smaller than the input dimensions. Hence, dimensionality reduction is achieved without losing much information.

B.2 Hierarchical Clustering Analysis

Hierarchical clustering analysis is a type of statistical data classification technique based on some kind of perceived similarity defined in the dataset. Clustering techniques [8] can be broadly categorized into *hierarchical* and *partitional* clustering. In hierarchical clustering, clusters can be chosen according to the linkage distance without defining the number of clusters *apriori*. In contrast, *partitional* clustering requires that the number of clusters be defined as an input to the process. Hierarchical clustering can be done in *agglomerative* (bottom-up) or *divisive* (top-down) manner. In the bottom-up approach, all of the points are defined as different clusters at the beginning. The most similar clusters are found and grouped together. The previous steps are repeated until all the data points are clustered. In *single linkage clustering*, similarity checking can be done by considering minimum distance among two different clusters. Alternatively, in *complete linkage clustering,* maximum distance among the two different clusters is chosen. *Average linkage clustering* refers to similarity checking based on mean distance between different clusters. The whole process of clustering produces a tree-like structure (dendrogram), where one axis represents different data points and the other represents linkage distance. The whole dataset can be categorized by selecting a particular *linkage distance*. Higher linkage distance expresses dissimilarity between the data points.

IV. Experimental Methodology

A. GPGPU-Sim and Simulator Configurations

In this study, we used GPGPU-Sim [6], a cycle accurate PTX-ISA simulator. GPGPU-Sim encapsulates a PTX ISA [5] functional simulator (CUDA-Sim) unit and a parallel SIMD execution performance simulator (GPU-Sim) unit. The simulator simulates shader cores, interconnection network, memory controllers, texture caches, constant caches, L1 caches, and off-chip DRAM. In the baseline configuration, different streaming multiprocessors (SM) are connected to the memory controllers through the mesh or crossbar interconnection network. On-chip memory controllers are connected to the off-chip memory chips. A thread scheduler distributes blocks among the shader cores in breadth-first manner; the least used SM is assigned the new block to achieve load balancing among cores [6]. The SIMD execution pipeline in the shader core has following stages: fetch, decode, pre-execute, execute, pre-memory (optional), memory access (texture, constant and other) and write-back. GPGPU-Sim provides L1 global and local memory caches. In the simulator compilation flow, *cudafe* [22] separates GPU code, which is further compiled by *nvopencc* [22] to produce PTX assembly. *Ptxas* assembler generates thread level shared memory, register and constant memory usage that is used by GPGPU-Sim during thread allocation in individual SM. GPGPU-Sim uses the actual PTX assembly instructions generated for functional simulation. Moreover, GPGPU-Sim implements a custom CUDA runtime library to divert the CUDA runtime API calls to GPGPU-Sim. Host C code is linked with the GPGPU-Sim custom runtime.

TABLE III
GPGPU-Sim Shader Configuration

	Conf 1	Conf 2	Base	Conf 3	Conf 4	Conf 5
Shader cores	8	8	28	64	110	110
Warp size	32	32	32	32	32	32
SIMD Width	32	32	32	32	32	32
DRAM controllers	8	8	8	8	11	11
DRAM queue size	32	32	32	32	32	128
Blocks / SM	8	2	8	8	8	16
Bus width	8bytes/cycle					
Const. / Texture cache	8KB / 64B (2-way, 64B line, LRU)					
Shared memory	16KB	8KB	16KB	16KB	16KB	32KB
Reg. count	8192	8192	16384	16384	16384	32768
Threads per SM	1024	512	1024	1024	1024	2048

TABLE IV
GPGPU-Sim Interconnect Configuration

	Conf 1	Conf 2	Base	Conf 3	Conf 4	Conf 5
Topology	Mesh	Mesh	Mesh	Crossbar	Mesh	Mesh
Routing	Dim. order	Dim. order	Dim. order	Dim. order	Dim. order	Dim. order
Virtual channels	2	2	2	1	4	4
VC buffers	4	4	4	8	16	16
Flit size	16 B	8 B	8 B	32 B	32 B	64 B

We have used the GPGPU-Sim configuration as specified in Tables III and IV. To show that the GPU configuration has no impact on the workload data, we have used 6 different configurations. Conf-2, baseline configuration, and Conf-5 are significantly different from each other, whereas other configurations are changed in a subtle manner to see the effect of few selected components. Additionally, we have heavily

instrumented the simulator to extract the GPU characteristics such as floating-point instruction count, arithmetic intensity, percentage of divergent branches, percentage of divergent warp, percentage of divergent branches in serialized section, length of serialized section, etc.

TABLE V
GPGPU WORKLOAD SYNOPSIS

Abbr.	Workload	Instr. Count	Arith. Intensity	Branch Div? / Merge Miss? / Shared Mem? / Barriers?
LV	Levenshtein Edit-distance calculation	32K	221.5	Y/Y/Y/N
BFS	Breadth First Search [29]	589K	99.7	Y/Y/Y/N
BP	Back Propagation [25]	1048K	167.6	Y/Y/Y/Y
BS	BlackScholes Option Pricing [28]	61K	1000	Y/Y/Y/N
CP	Columbic Potential [26]	8K	471.8	Y/Y/Y/N
CS	Separable Convolution [28]	10K	3.1	Y/Y/Y/Y
FWT	Fast Walsh Transform [28]	32K	289.1	Y/Y/Y/Y
GS	Gaussian Elimination [25]	921K	5.7	Y/Y/Y/Y
HS	Hot Spot [31]	432K	101.4	Y/Y/Y/Y
LIB	LIBOR [32]	8K	353.6	Y/Y/Y/N
LPS	3D Laplace Solver [30]	12K	118.9	Y/Y/Y/Y
MM	Matrix Multiplication [28]	10K	42.4	Y/Y/Y/Y
MT	Matrix Transpose [28]	65K	195.6	Y/Y/Y/Y
NN	Neural Network [34]	66K	71.5	Y/Y/Y/N
NQU	N-Queen Solver [35]	24K	209.3	Y/Y/Y/Y
NW	Needleman Wunsch [25]	64	80.1	Y/Y/Y/Y
PF	Path Finder [25]	1K	378.1	Y/Y/Y/Y
PNS	Petri Net Simulation [26]	2.5K	411.5	Y/Y/Y/Y
PR	Parallel Reduction [28]	830K	1.0	Y/Y/Y/Y
RAY	Ray Trace [36]	65K	335.0	Y/Y/Y/Y
SAD	Sum of Absolute Difference [26]	11K	67.7	Y/Y/Y/Y
SP	Scalar Product [28]	32K	284.9	Y/Y/Y/Y
SRAD	Speckle Reducing Anisotropic Diffusion [26]	460K	14.0	Y/Y/Y/Y
STO	Store GPU [37]	49K	361.9	Y/Y/Y/N
CL	Cell [25]	64	765.2	Y/Y/Y/Y
HY	Hybrid Sort [38]	541K	32.1	Y/Y/Y/Y
KM	K-Means [39]	1497K	20.2	Y/Y/Y/N
MUM	MUMmerGPU [40]	50K	468.5	Y/Y/Y/N
NE	Nearest Neighbor [25]	60K	196.8	Y/Y/Y/N
BN	Binomial Options [28]	131K	39.4	Y/Y/Y/Y
MRIF	Magnetic Resonance Imaging FHD [44]	1050K	86.7	Y/Y/Y/N
MRIQ	Magnetic Resonance Imaging Q [26]	526K	107.6	Y/Y/Y/N
DG	Galerkin time-domain solver [41]	1035K	11.4	Y/Y/Y/Y
SLA	Scan of Large Arrays [28]	1310K	1.9	Y/Y/Y/Y
SS	Similarity Score [25]	51K	1.1	Y/Y/Y/Y
AES	AES encryption [42]	65K	70.25	N/Y/Y/Y
WP	Weather Prediction [43]	4.6K	459.8	Y/Y/Y/N
64H	64 bin histogram [28]	2878K	22.8	Y/Y/Y/Y

B. GPGPU Workloads

To perform GPU benchmark characterization analysis, we have collected a large set of available GPU workloads from *Nvidia CUDA SDK* [28], *Rodinia* Benchmark [25], *Parboil* Benchmark [26] and some third party applications. Due to simulation issues (e.g. simulator deadlock), we have excluded some benchmarks from our analysis. The problem size of the workloads is scaled to avoid long simulation time or unrealistically small workload stress. Table V lists the basic workload characteristics.

TABLE VI
GPGPU WORKLOAD EVALUATION METRICS

Evaluation metric	Synopsys
Activity factor	Average percentage of threads active at a given time.
SIMD parallelism	Speedup with infinite number of SP per SM.
DRAM efficiency	% of time spent sending the data across the pins of DRAM when other commands are pending /serviced.

C. Workload Evaluation Metrics

To evaluate the accuracy and the effectiveness of the proposed workload characteristics, we use the set of metrics as listed in Table VI. *Activity factor* [24] is defined as the average number of active threads at a given time during the execution phase. Several branch divergence related characteristics of the benchmark change this parameter. For example, the absence of branch divergence produces an *activity factor* of 100%. *SIMD parallelism* [24] captures the scalability of a workload. Higher value for *SIMD parallelism* indicates that the workload performance will improve on a GPU that have higher SIMD width. *DRAM efficiency* [6] describes how frequently memory accesses are requested during kernel computation. It also captures the amount of time spent to perform DRAM memory transfer during overall kernel execution. If a benchmark has large number of shared memory accesses or the benchmark has ALU operations properly balanced in between memory operations, then the metric will show higher value.

D. Experiment Stages

The experimental procedure consists of following steps:

(1) The heavily instrumented GPGPU-Sim simulator retrieves the statistics of GPU characteristics listed in Tables I, II and VI for all the configurations described in Tables III and IV. Six different configurations are simulated to demonstrate the microarchitectural independence of the GPGPU workload characteristics.

(2) We have performed vectorization of some characteristics to produce 10-bin or 5-bin histograms from the GPGPU-Sim output. This process provides an input matrix for principal component analysis of size 129×38 (dimensions×workloads). Few dimensions of the matrix with zero *standard deviation* are kept out of the analysis to produce a 93×38 normalized data matrix.

(3) PCA is performed using STATISTICA [45]. Several PCAs are performed on the whole workload set according to different workload characteristics subspaces and all the characteristics.

(4) Based on required percentage of total variance, p principal components are chosen.

(5) STATISTICA is used to perform hierarchical cluster analysis. Both single linkage and complete linkage based dendrograms are generated using m principal components, where $m < p$ (p: total number of PC retained).

V. RESULTS AND ANALYSIS

A. Microarchitecture Impact on GPGPU Workload Characteristics

We verify that GPGPU workload characteristics are independent of the underlying microarchitecture. If the microarchitecture has little or no impact on the set of benchmark characteristics, then benchmarks executed on different microarchitectures will be placed close to each other in uncorrelated principal component domain. Figures 2 and 3 show the dendrogram of all the benchmarks and the distribution of the variance obtained in different PCs. According to

Kaiser Criterion [46] we need to consider all the principal components till PC-20. Yet we have decided to consider first 7 principal components that retain 70% of the total variance and first 16 principal components that retain 90% of the total variance. In Figure 3 we see that executions of same benchmark on different microarchitectures are having small linkage distance, which demonstrates strong clustering. This suggests that irrespective of the varied microarchitecture configurations (e.g. shader cores, register file size, shared memory size, interconnection network configuration etc.), our proposed metrics are capable of capturing GPGPU workload characteristics. Note that, in Figure 3 some benchmarks' execution on a particular microarchitecture is missing due to lack of microarchitectural resources of the GPU or simulation issues. For example, Conf-2 (Tables III and IV) is incapable of executing benchmark-problem size pair of STO, NQU, HY and WP.

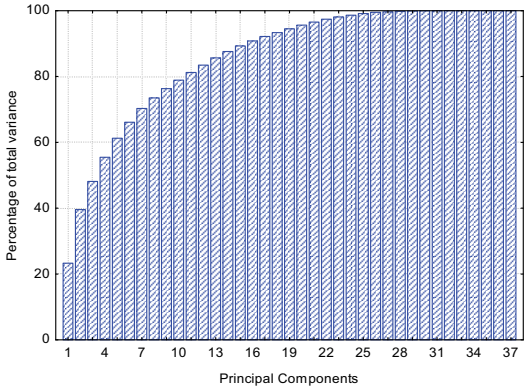


Fig. 2. Cumulative Distribution of Variance by PC

B. GPGPU Workload Classification

The program-input pairs defined as workload in Table V show significantly distinct behavior in PCA domain. The hierarchically clustered principal components are listed in Figures 4, 5, 6, and 7. To avoid the hierarchical clustering artifacts, we have done both single linkage and complete linkage clustering dendrograms [27]. We have observed that for the set of 4 and 6 workloads the single linkage and complete linkage clustering shows identical results irrespective of 70% variance and 90% variance cases. Benchmarks in set of 4 are SS, SLA, PR and NE. The set of 6 includes SS, SLA, PR, BN, KM and NE. For set size of 8 and 12 we have found single linkage and complete linkage clustering deviates by 1 and 2 workloads respectively, for the 90% variance case. Table VII depicts results; bold type face highlights the differences in single and complete linkage clustering. To represent a cluster we choose the closest benchmark to the centroid of that cluster. The results for 70% of total variance are presented in Figures 6 and 7. Dendrogram also highlights that *Rodinia*, *Parboil* and *Nvidia CUDA SDK* benchmark suites demonstrate the diversity in decreasing order according to the available workloads. Table VII shows the time savings obtained for different sets in the form of speedup. Out of all the benchmarks SS and 64H contribute 26% and 29% of the total simulation time respectively. Speedup numbers are in fact higher without SS. SS demonstrates very distinct characteristics from the rest; such as high instruction count, low arithmetic intensity, very high kernel count, diverse inter-warp coalescing behavior, branch divergence diversity, different types of kernels and diverse instruction mix. As expected, overall speedup decreases as we increase the set size. Due to the inclusion of 64H, the set of 12 in complete linkage clustering does not show high speedup. Architects can choose the set size to achieve the trade-off between available simulation time and amount of accuracy desired.

C. GPU Microarchitecture Evaluation

Figure 8 shows the performance comparison of different evaluation metrics as GPU microarchitecture varies. Table VIII shows the average error observed. Due to lack of computing resources and simulation issues with some benchmarks (e.g. SS, STO, HY, NQU), configurations such as Conf-2, Conf-4 and Conf-5 are not considered. This should not affect the results as the benchmark characteristics are microarchitecture agnostic. Maximum average error for activity factor is less than 17%. It reveals that the subsets are capable of representing branch divergence characteristics present in the whole set. Divergence based clustering shows branch divergence behavior is diverse. Therefore, a small set of 4 is relatively less accurate in capturing divergence behavior. As we increase the set size to 6 and 8, we reach the average error as close as 1%. Average error in SIMD parallelism suggests that we are capturing the kernel level parallelism through activity factor and dynamic instruction count with maximum error less than 18%. Increase of set size also decreases the average error for SIMD parallelism. DRAM efficiency shows that coalescing characteristics, memory request and memory transfer behaviors are captured closely with maximum error less than 6%. We also observe that the increase in subset size decreases the average error.

D. GPGPU Workload Fundamentals Analysis

In this subsection we analyze the workloads from the point of view of input characteristics. Architects are often interested in identifying the most influential characteristics to tune the performance of the workload on the microarchitecture. Different PCs are arranged in decreasing order of variances. Hence, by observing their factor loadings we can infer the impact of the input characteristics. We choose to analyze top 4 principal components, which account for 53% of the total variance.

Figure 9 shows the factor loadings for principal component 1 and 2, which account for 39% of total variance. Figure 10 shows the scatter plot. SS, SLA, PR, GS, SRAD, 64H, HS, and KM show relatively low arithmetic intensity than others. SS, SLA, PR, GS, SRAD, 64H, HS, and KM have large number of threads spawned, small number of instructions in a kernel, moderate merge miss count, and good row locality. The exceptions are CS and HY, which possess large kernel count. Moreover, workloads excluding these are relatively less diverse in the previously mentioned behavioral domain. Close examination of the simulation statistics verify that benchmarks except SS, SLA, PR, GS, SRAD, 64H, and KM have good arithmetic intensity, large number of kernels and kernel level diversity. There are no benchmarks with very high arithmetic intensity. SS exhibits high integer instruction count, host to device data transfer, low branch divergence, and per thread shared memory usage. In contrast, SLA demonstrates low merge miss, good row locality and branch divergence. Simulation statistics verifies observed branch divergence, merge miss, and row locality.

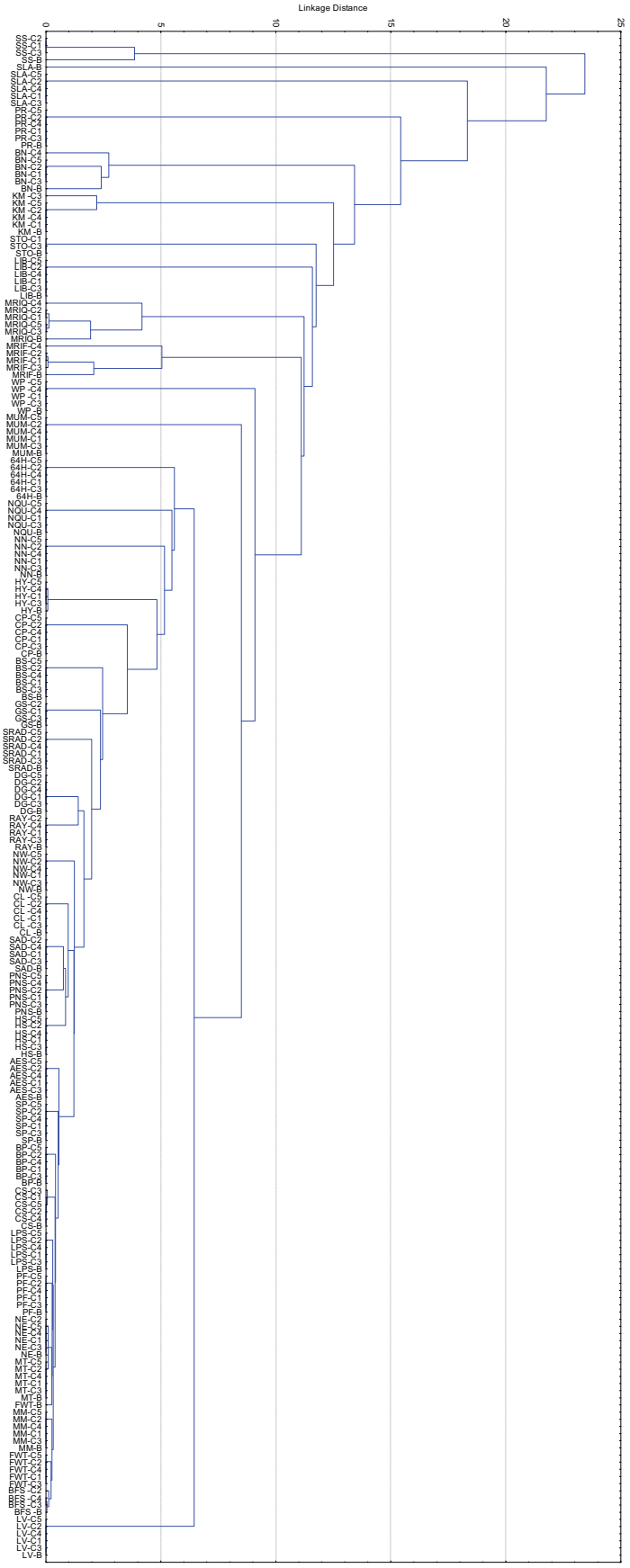


Fig. 3. Dendrogram of all Workloads

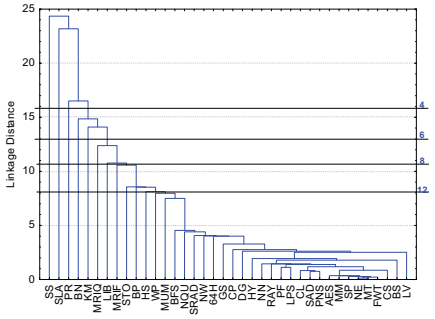


Fig. 4. Dendrogram of Workloads (SL, 90% Var)

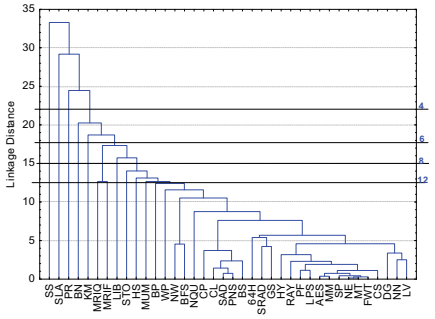


Fig. 5. Dendrogram of Workloads (CL, 90% Var)

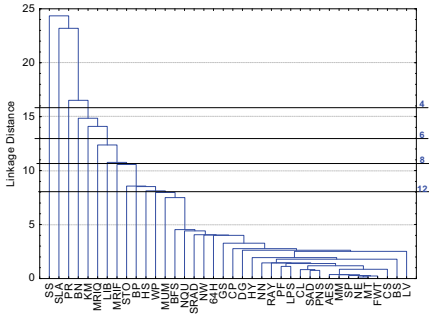


Fig. 6. Dendrogram of Workloads (SL, 70% Var)

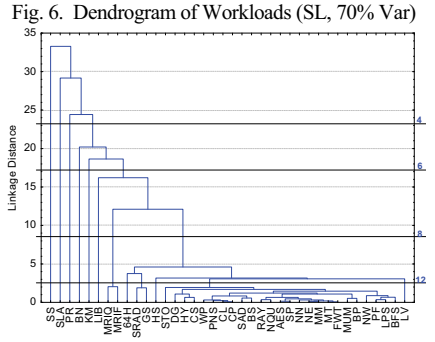


Fig. 7. Dendrogram of Workloads (CL, 70% Var)

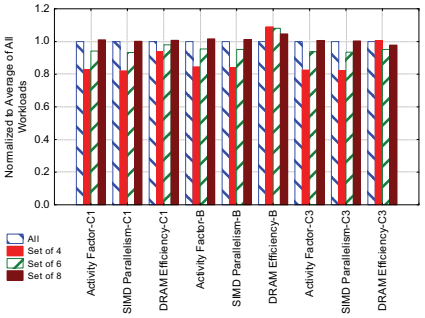


Fig. 8. Performance comparison (normalized to total workloads) of different evaluation metrics across different GPU Microarchitectures (C1: Config. 1, B: Baseline, C3: Config. 3)

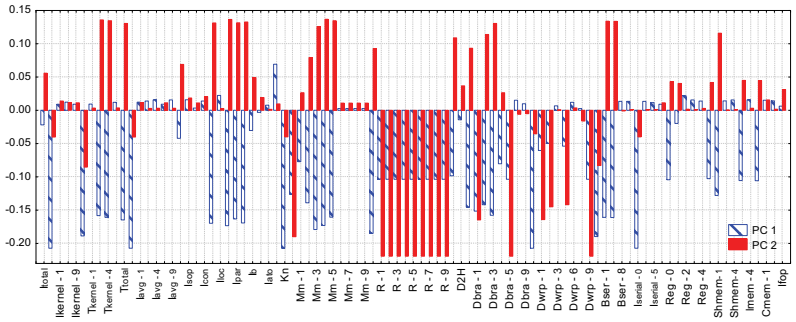


Fig. 9. Factor Loading of PC-1 and PC-2

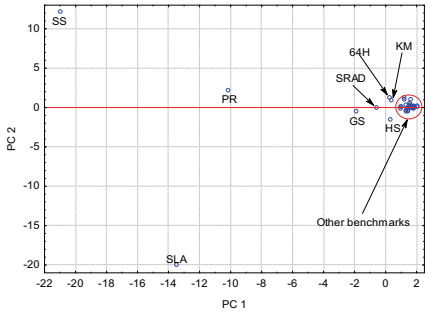


Fig. 10. PC-1 vs. PC-2 Scatter Plot

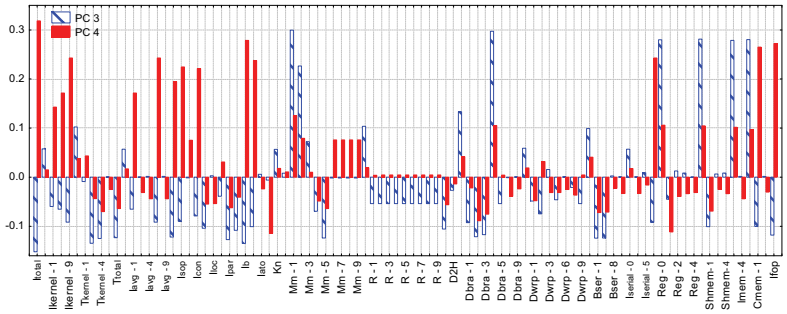


Fig. 11. Factor Loading of PC-3 and PC-4

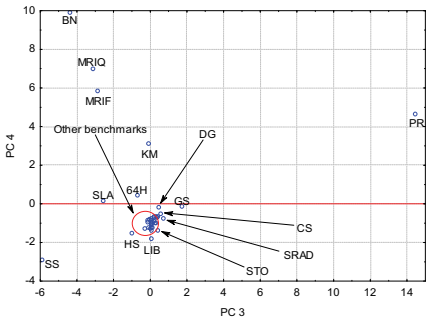


Fig. 12. PC-3 vs. PC-4 Scatter Plot

TABLE VII
GPGPU WORKLOAD SUBSET FOR 90% VARIANCE

	Single linkage (SL)	SL speedup	Complete linkage (CL)	CL speedup
Set 4	SS, SLA, PR, NE	3.60	SS, SLA, PR, NE	3.60
Set 6	SS, SLA, PR, KM, BN, NE	2.38	SS, SLA, PR, KM, BN, NE	2.38
Set 8	SS, SLA, PR, KM, BN, LIB, MRIQ, **NE**	1.80	SS, SLA, PR, KM, BN, LIB, MRIQ, **MRIF**	1.53
Set 12	SS, SLA, PR, KM, BN, LIB, MRIQ, MRIF, STO, HS, **BP**, **NE**	1.51	SS, SLA, PR, KM, BN, LIB, MRIQ, MRIF, STO, HS, **64H**, **MUM**	1.05

TABLE VIII
AVERAGE % OF ERROR AS MICROARCHITECTURE VARIES

Evaluation metric	Set 4	Set 6	Set 8
Activity Factor	16.7	5.5	1.1
SIMD parallelism	17.2	6.0	0.5
DRAM efficiency	5.2	5.0	2.5
Average	13.1	5.5	1.4

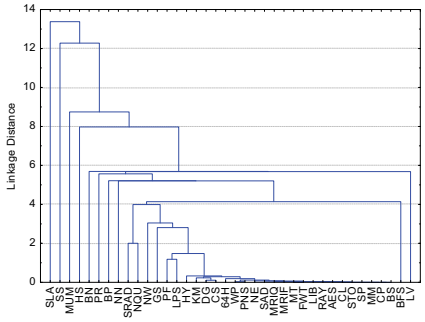


Fig. 13. Dendrogram based on Divergence Characteristics

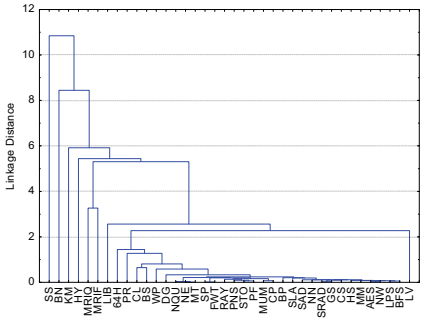


Fig. 14. Dendrogram based on Instruction Mix

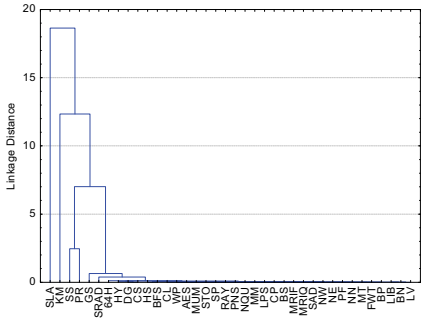


Fig. 15. Dendrogram based on Merge Miss and Row Locality

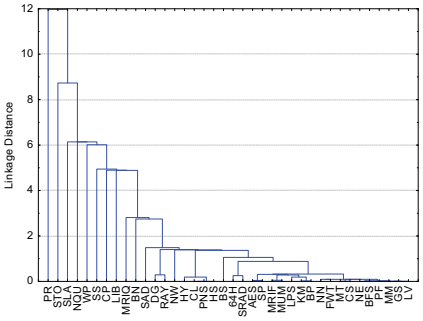


Fig. 16. Dendrogram based on Kernel Characteristics

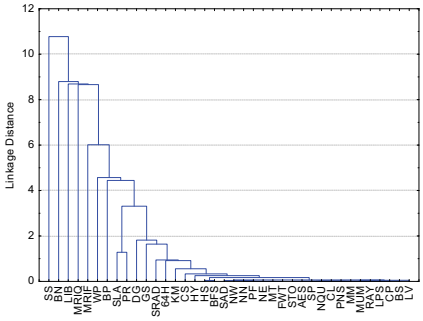


Fig. 17. Dendrogram based on Kernel Stress

Figures 11 and 12 show the PC-3 vs. PC-4 (14% of total variance) factor loadings and scatter plot. PC-3 shows that PR has large number of kernels, low merge miss and moderate branch divergence. BN, SLA, MRIF and MRIQ possess moderate branch divergence and heavyweight threads. BN, MRIQ, MRIF, KM, and PR program-input pairs have long serialized section due to branch divergence (large I_{serial}), high barrier and branch instruction count, heavy weight threads and more computations.

E. Characteristics Based Classification

In Section 3.1 we have mentioned several subspaces, such as instruction mix subspace, coalescing characteristics subspace, divergence characteristics subspace, to categorize the workload characteristics. In Figures 13, 14, 15, 16 and 17, we present dendrograms generated from the principal components (90% of total variance) obtained from PCA analysis of the workloads based on those different subspaces. Architects interested in improving branch divergence hardware might want to simulate SS, SLA, MUM, HS, BN and NE as they exhibit relatively large amount of branch divergence diversity. On the other hand, SS, BN, KM, MRIF, MRIQ, and, HY have distinct behavior in terms of instruction mix. These are valuable benchmarks for evaluating the effectiveness of design with ISA optimization. Interestingly, workloads excluding SLA, KM, SS and PR show similar types of inter-warp and intra-warp memory coalescing behavior. Hence, SLA, KM, SS and PR provide diversity to evaluate microarchitecture level memory access optimizations. Though it is possible that the workloads with similar inter and intra warp coalescing have large variations in cache behavior when the size of the cache is increasing beyond that used to measure intra-warp coalescing. Figure 16 shows the static kernel characteristics of different benchmarks. PR, STO, SLA, NQU, WP, SS, CP and LIB show distinct kernel characteristics. SS, BN, LIB, MRIF, MRIQ, WP, BP and NE demonstrate diverse behavior in terms of instructions executed by each thread, total thread count, total instruction count etc.

F. Discussions and Limitations

This research explores PTX translated GPGPU workload diversity in terms of different microarchitecture independent program characteristics. The GPU design used in our research closely resembles Nvidia GPUs. However, growing GPU industry has several other microarchitecture designs (AMD-ATI [23], Intel Larrabee [50]), programming models (ATI Stream [23], OpenCL[49]) and ISAs (ATI intermediate language, x86, Nvidia Native Device ISA, ATI Native Device ISA). The microarchitecture independent characteristics proposed in this paper are by and large applicable to ATI GPUs though they use different virtual ISA (ATI IL). For example, SFU instructions may be specific to Nvidia, but AMD-ATI also processes these instructions in a transcendental unit inside the thread processor. Branch divergence and memory coalescing behavior of the kernels are a characteristic of the workload and do not affect the results produced in this paper for ATI GPUs. However, for Intel Larrabee architecture, these characteristics will be different because of dissimilarity in the programming model. Also, Larrabee microarchitecture is different from traditional GPUs. Hence, for Larrabee the suitability of using the proposed metrics to represent workload characteristics needs further exploration.

PTX ISA changes are minor over the generations; therefore it has insignificant effect on the results. In addition, PTX instruction mapping to different generations of native Nvidia ISAs and programming model change has little impact on the GPGPU kernel characteristics because we characterize the kernels in terms of the PTX virtual ISA.

VI. Related Work

Saavedra et al. [47] demonstrated how workload performance on a new microarchitecture can be predicted using the study of microarchitecture independent and dependent workload characteristics. However, their work did not consider the correlated nature of different characteristics. In [11, 48], Eeckhout et al. demonstrated a technique to reduce the simulation time while keeping the benchmark diversity information intact by performing PCA and clustering analysis on correlated microarchitecture independent workload characteristics. [12] showed that the 26 CPU2000 workloads only stress four different bottlenecks by collecting data from 340 different machines. [9] reported the redundancy in SPEC2006 benchmarks using the same technique. Results showed that 6 CINT2006 and 8 CFP2006 benchmarks can be representative of the whole workload set. In previously mentioned works, the following characteristics of the program were widely adopted: instruction mix, branch prediction, instruction level parallelism, cache miss rates, sequential flow breaks etc. The using of microarchitecture and transactional architecture independent characteristics, such as transaction percentage, transaction size, read/write set size ratio and conflict density, for clustering transactional workloads was performed in [10]. Our approach differs from all the above as we characterize the data parallel GPGPU workloads using several newly proposed microarchitecture independent characteristics.

So far, there have been very few studies [24, 25] on GPGPU workload classification. [24] characterized 50 different PTX kernels which include *NVIDIA CUDA SDK* kernels and *Parboil* benchmarks suit. This study does not consider correlation among various workload characterization metrics. In contrast, we use a standard statistical methodology with a wide range of workload characterization metrics. Moreover, [24] have not considered diverse benchmark suites like *Rodinia* and *Mars* [33]. In [25], the *MICA framework* [11] was used to characterize single-core CPU version of the GPGPU kernels, which fails to capture branch divergence, row access locality, and memory coalescing, the characteristics of numerous parallel threads running on massively parallel GPU microarchitecture. Moreover, instruction level parallelism has less effect in GPU performance due to the simplicity of the processor core architecture. More than data stream size, data access pattern plays important role in boosting the application performance on GPU. We capture these over-looked features in our program characteristics to look into the GPGPU workload behavior in detail.

VII. Conclusions

The emerging GPGPU workload space has not been explored methodically to understand the fundamentals of the workloads. To understand the GPU workloads, we have proposed a set of microarchitecture independent GPGPU workload characteristics that are capable of capturing five important behaviors: kernel stress,

kernel characteristics, divergence characteristics, instruction mix, and coalescing characteristics. We have also demonstrated that the proposed evaluation metrics accurately represents the input characteristics. This work provides GPU architect a clear understanding of the available GPGPU workload space to design better GPU microarchitecture. Our endeavor also demonstrates that workload space is not properly balanced with available benchmarks; while SS, SLA, PR workloads show significantly different behavior due to their large number of diverse kernels, the rest of the workloads provide similar characteristics. We also observe that benchmark suites like *Nvidia CUDA SDK*, *Parboil*, and *Rodinia* has different behavior in terms workload space diversity. Our research shows that branch divergence diversity is best captured by SS, SLA, MUM, HS, BN, and, NE. SLA, KM, SS, and, PR show relatively large memory coalescing behavior diversity than the rest. SS, BN, KM, MRIF, and HY have distinct instruction mix. Kernel characteristic is diverse in PR, STO, NQU, SLA, WP, SS, CP, and LIB. We also show that among the chosen benchmarks divergence characteristics are most diverse and coalescing characteristic are least diverse. Therefore, we need benchmarks with more diverse memory coalescing behavior. In addition, we show that simulation speedup of 3.5× can be achieved by removing the redundant benchmarks.

ACKNOWLEDGEMENTS

This work is supported in part by NSF grants CNS-0834288, CCF-0845721 (CAREER), SRC grant 2008-HJ-1798, and by three IBM Faculty Awards. The authors acknowledge the UF HPC Center for providing computational resources. We also acknowledge anonymous reviewers for their valuable suggestions.

REFERENCES

[1] G. Yuan, A. Bakhoda, and T. Aamodt, Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures, MICRO, 2009.
[2] W. Fung, I. Sham, G. Yuan, and T. Aamodt, Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow, MICRO, 2007.
[3] T. Halfhill, Looking Beyond Graphics, Microprocessor Report, 2009.
[4] D. Tarjan, J. Meng, and K. Skadron, Increasing Memory Miss Tolerance for SIMD Cores, SC, 2009.
[5] NVIDIA Compute PTX: Parallel Thread Execution ISA, Version 1.4, 2009.
[6] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, Analyzing CUDA Workloads using a Detailed GPU Simulator, ISPASS, 2009.
[7] G. Dunteman, Principal Components Analysis, SAGE Publications, 1989.
[8] C. Romesburg, Cluster Analysis for Researchers, Lifetime Learning Publications, 1984.
[9] A. Joshi, A. Phansalkar, L. Eeckhout, and L. John, Measuring Benchmark Similarity using Inherent Program Characteristics, IEEE Transactions on Computers, Vol.55, No.6, 2006.
[10] C. Hughe, J. Poe, A. Qouneh, and T. Li, On the (Dis)similarity of Transactional Memory Workloads, IISWC, 2009.
[11] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, Designing Computer Architecture Research Workloads, Computer, Vol. 36, No. 2, 2003.
[12] H. Vandierendonck and K. Bosschere, Many Benchmarks Stress the Same Bottlenecks, Workshop on Computer Architecture Evaluation Using Commercial Workloads, 2004.
[13] J. Henning, SPEC CPU2000: Measuring CPU Performance in the New Millennium, IEEE Computer, pp. 28-35, July 2000.
[14] C. Lee, M. Potkonjak, and W. Smith, MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems, MICRO, 1997.
[15] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, MiBench: A Free, Commercially Representative Embedded Benchmark Suite, Workshop on Workload Characterization, 2001.
[16] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, ISCA, 1995.
[17] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, STAMP: Stanford Transactional Memory Applications for Multi-Processing, IISWC, 2008.
[18] C. Bienia, S. Kumar, J. Singh, and K. Li, The PARSEC Benchmark Suite: Characterization and Architectural Implications, Princeton University Technical Report, 2008.
[19] NVIDIA CUDA™ Programming Guide Version 2.3.1, Nvidia Corporation, 2009.
[20] J. Nickolls, I. Buck, M. Garland, and K. Skadron, Scalable Parallel Programming with CUDA, Queue 6, 2 (Mar. 2008), 40-53.
[21] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, NVIDIA Tesla: A Unified Graphics and Computing Architecture, Micro, vol.28, no.2, 2008.
[22] The CUDA Compiler Driver NVCC, Nvidia Corporation, 2008.
[23] Technical Overview, ATI Stream Computing, AMD Inc, 2009.
[24] A. Kerr, G. Diamos, and S. Yalamanchilli, A Characterization and Analysis of PTX Kernels, IISWC 2009.
[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing, IISWC, 2009.
[26] Parboil Benchmark suite. URL: http://impact.crhc.illinois.edu/ parboil.php.
[27] M. Hughes, Exploration and Play Re-visited: A Hierarchical Analysis, International Journal of Behavioral Development, 1979.
[28] http://www.nvidia.com/object/cuda_sdks.html
[29] P. Harish and P. J. Narayanan, Accelerating Large Graph Algorithms on the GPU Using CUDA, HiPC, 2007.
[30] M. Giles, Jacobi Iteration for a Laplace Discretisation on a 3D Structured Grid, http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/laplace3d.pdf, April, 2008.
[31] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, and K. Skadron, M. R. Stan, Hotspot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design, IEEE Transactions on VLSI Systems 14 (5) (2006).
[32] M. Giles and S. Xiaoke, Notes on using the NVIDIA 8800 GTX graphics card. http://people.maths.ox.ac.uk/~gilesm/hpc/.
[33] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, Mars: a MapReduce Framework on Graphics Processors, PACT, 2008.
[34] Billconan and Kavinguy, A Neural Network on GPU. http://www.codeproject.com/KB/graphics/GPUNN.aspx.
[35] Pcchen. N-Queens Solver, http://forums.nvidia.com/index.php?showtopic=76893, 2008.
[36] Maxime. Ray tracing. http://www.nvidia.com/cuda.
[37] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, StoreGPU: Exploiting Graphics Processing Units to accelerate Distributed Storage Systems, HPDC, 2008.
[38] E.Sintorn and U. Assarsson, Fast Parallel GPU-sorting using a Hybrid Algorithm, Journal of Parallel and Distributed Computing, Volume 68, Issue 10, October 2008.
[39] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, J. Pisharath, G. Memik, and A. Choudhary, MineBench: A Benchmark Suite for Data Mining Workloads, IISWC, 2006.
[40] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, High-throughput Sequence Alignment using Graphics Processing Units, BMC Bioinformatics, 8(1): 474, 2007.
[41] T. Warburton, Mini Discontinuous Galerkin Solvers. http://www.caam.rice.edu/~timwar/RMMC/MIDG.html, 2008.
[42] S. Manavski, CUDA compatible GPU as an Efficient Hardware Accelerator for AES Cryptography, ICSPC, 2007.
[43] J. Michalakes and M. Vachharajani, GPU Acceleration of Numerical Weather Prediction, IPDPS, 2008.
[44] S. Stone, J. Haldar, S. Tsao, W. Hwu, Z. Liang, and B. Sutton, Accelerating Advanced MRI Reconstructions on GPUs, Computing Frontiers, 2008.
[45] StatSoft, Inc. STATISTICA, http://www.statsoft.com/.
[46] K. Yeomans and P.Golder, The Guttman-Kaiser Criterion as a Predictor of the Number of Common Factors", Journal of the Royal Statistical Society. Series D (The Statistician), Vol. 31, No. 3, 1982.
[47] R. H. Saavedra and A. J. Smith, Analysis of Benchmark Characteristics and Benchmark Performance Prediction, ACM Trans. Computer Systems, 1998.
[48] K. Hoste and L. Eeckhout. Microarchitecture-independent Workload Characterization. IEEE Micro, 27(3):63–72, 2007.
[49] http://www.khronos.org/opencl/
[50] L. Seiler, D. Carmean, E. Sprangle, T.Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, Larrabee: A Many-core X86 Architecture for Visual Computing, ACM Trans. Graph. 27-3, 2008.
[51] A. Phansalkar, A. Joshi, and L. John, Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite, ISCA, 2007.

A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads

Shuai Che
sc5nf@virginia.edu

Jeremy W. Sheaffer
jws9c@cs.virginia.edu

Michael Boyer
mwb7w@cs.virginia.edu

Lukasz G. Szafaryn
lgs9a@virginia.edu

Liang Wang
lw2aw@virginia.edu

Kevin Skadron
skadron@cs.virginia.edu

The University of Virginia
Department of Computer Science

***Abstract*—The recently released Rodinia benchmark suite enables users to evaluate heterogeneous systems including both accelerators, such as GPUs, and multicore CPUs. As Rodinia sees higher levels of acceptance, it becomes important that researchers understand this new set of benchmarks, especially in how they differ from previous work. In this paper, we present recent extensions to Rodinia and conduct a detailed characterization of the Rodinia benchmarks (including performance results on an NVIDIA GeForce GTX480, the first product released based on the Fermi architecture). We also compare and contrast Rodinia with Parsec to gain insights into the similarities and differences of the two benchmark collections; we apply principal component analysis to analyze the application space coverage of the two suites. Our analysis shows that many of the workloads in Rodinia and Parsec are complementary, capturing different aspects of certain performance metrics.**

I. Introduction

Computer systems are increasingly exposing a heterogeneous computing model consisting of accelerators—such as graphics processors (GPUs), media processors, and even reconfigurable hardware like FPGAs—combined with one or more conventional CPUs. GPUs, for instance, offer parallelism at scales unachievable with other processors and afford about an order of magnitude greater peak throughput than general-purpose, multicore CPUs, while the CPUs offer high single-thread performance and programmability.

A vision of heterogeneous computer systems that incorporate diverse accelerators and automatically select the best computational unit for a particular task is widely shared among researchers and many industry analysts; however, there are no agreed-upon benchmarks to support the research needed in the development of such a platform. There are many benchmark suites for parallel computing on general-purpose CPU architectures, but accelerators fall into a gap that is not covered by current benchmark suites or benchmark development. There is a dearth of publicly available code for heterogeneous platforms.

The Rodinia benchmark suite [8], a set of free and open benchmarks and associated methodologies, was developed to address these concerns. The Rodinia applications are designed for *heterogeneous* computing infrastructures, and, using OpenMP and CUDA, target both GPUs and multicore CPUs. The implementations for each distinct platform can also serve as independent suites to evaluate multicore and manycore architectures separately. The Rodinia suite is structured to span a range of parallelism and compute patterns, providing researchers with various feature options to identify architectural bottlenecks and to fine tune hardware designs.

Several multithreaded benchmark suites for multicore CPUs, including SPLASH-2 [35], Parsec [5], and SPEC OMP [29], are available. Rodinia was developed to address the issues of benchmarking heterogeneous systems, particularly those including a GPU. There is growing support for use of the Rodinia workloads [6], [8]–[10], [24], but there are some important questions yet to be answered:

- How much do those Rodinia workloads which are designed for heterogeneous platforms (those with GPU accelerators) differ from those of other suites designed for multicore CPUs?
- Do the workload designs of other suites demonstrate overlapping or orthogonal features?
- How well do the chosen applications span the workload space?
- How well can traditional, multithreaded CPU workloads map onto GPU platforms?

A better understanding of these issues will not only expand the knowledge of parallel benchmark construction, but could also inform decisions on workload scheduling and partitioning on different architectures and guide researchers to choose appropriate benchmarks for their research as well.

In this paper we make the following contributions:

- We present important extensions to the Rodinia benchmark suite that have been added since its initial publication at IISWC 2009 [8].
- We conduct a more detailed characterization of the Rodinia GPU workloads to aid researchers in understanding

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

the characteristics of Rodinia.
- We evaluate the Rodinia benchmarks on a recently released NVIDIA GTX480, which is based on the Fermi architecture with traditional L1 and L2 caches, identifying some bottlenecks of the new GPU architecture.
- We perform an application space study, comparing the multithreaded CPU implementations of Rodinia with those of Parsec, and evaluate the extent to which the program selections of the two suites overlap.
- We present analysis and discussion of important, open research topics, including the need for new parallel performance metrics, for an effective application taxonomy, and for a general application space study of multithreaded workloads, and we discuss the challenges that make porting existing suites difficult.

II. Overview of Rodinia

As apposed to Parsec and SPLASH-2, which target homogeneous platforms, Rodinia workloads are selected and designed for heterogeneous computing platforms including both CPUs and devices such as GPUs and FPGAs [10]. Rodinia not only covers applications from emerging domains such as bioinformatics, data mining, and image processing, but also includes the accelerator implementations of important, classical algorithms like LU decomposition and graph traversal. The Berkeley Dwarf taxonomy [1] was initially used as a guideline to choose applications for Rodinia in order to avoid missing important parallel patterns. Table I illustrates the Rodinia applications and their corresponding domains and Dwarves.

The Rodinia benchmarks are currently implemented in OpenMP and CUDA. As OpenCL provides an attractive alternative to CUDA, we are producing OpenCL ports, as well; these are not complete, nor as mature as the OpenMP and CUDA implementations. OpenCL and CUDA use very similar sets of abstractions, such that CUDA is sufficient for the characterization and diversity analysis presented in this paper. We expect that our reported results will transfer directly to the OpenCL ports when they are complete.

Rodinia has some important features that differentiate it from other benchmark suites:

- Rodinia implementations take advantage of non-traditional memory hierarchies, like scratchpad and texture units, for general purpose computation. Cell and ClearSpeed are two examples in a trend to use other types of memories as alternatives to hardware-managed cache. This trend in turn requires benchmark development to keep up with such an evolution.
- Rodinia provides multiple versions of some applications, with successive layers of optimization, allowing designers to evaluate the impact of multiple different implementations on their architecture or compiler designs.
- Rodinia's applications adopt an "offloading" model which assumes that accelerators use a memory space disjoint from main memory.
- Rodinia provides a set of applications from which it may be relatively hard for compilers to automatically generate accelerator code.

A. Rodinia Extensions

Since the release of the first version of Rodinia [8], we have been adding applications to enrich the workload set. The newly added applications include *Heartwall-Tracking*, *LU Decomposition*, *MUMmer* and *Computational Fluid Dynamics*. The major criterion applied in selecting these applications was their use of advanced data structures. These data structures allow the applications to demonstrate new types of parallelism and inter-thread communications not seen in other members of the Rodinia suite.

LU Decomposition (LUD): LU Decomposition is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. This application has many row-wise and column-wise interdependencies and requires significant optimization to achieve good parallel performance.

Motivation for inclusion: LU Decomposition exhibits significant inter-thread sharing and row and column dependencies.

Heartwall Tracking (HW): The Heart Wall [31] application tracks the changing shape of the walls of a mouse heart over a sequence of 104 ultrasound images, each with a resolution of 609×590 pixels. In its initial stage, the program performs several image processing passes—edge detection, SRAD despeckling (part of Rodinia), morphological transformation, and dilation—on the first image in the sequence in order to detect partial shapes of inner and outer heart walls. To reconstruct approximated, full shapes of heart walls for tracking purposes, the application generates ellipses that are superimposed over the image and sampled to mark points on the heart walls. In its final stage, the program tracks the changing shapes of the two heart walls by detecting the movement of certain sample points throughout the sequence of images.

Motivation for inclusion: Heartwall Tracking presents a pattern of *braided parallelism*—a mixture of data and task parallelism—which is absent from other Rodinia benchmarks. The application is coarsely parallelized according to independent tasks (TLP); each task is then finely parallelized according to independent data operations (DLP). The processing of a frame is implemented as a single GPU kernel in order to successfully implement braided parallelism and avoid kernel launch overhead. This structure requires the inclusion of some non-parallel computation into the kernel, leading to a slight warp under-utilization but overall greater performance.

Computational Fluid Dynamics (CFD): The CFD solver is an unstructured-grid, finite-volume solver for the three-dimensional Euler equations for compressible flow. Effective GPU memory bandwidth is improved by reducing total global memory accesses and overlapping redundant computation, as well as by using an appropriate numbering scheme and data

TABLE I
RODINIA APPLICATIONS AND KERNELS ('*' DENOTES KERNEL).

Application	Dwarf	Domain	Problem Sizes
Kmeans	Dense Linear Algebra	Data Mining	204800 data points, 34 features
Needleman-Wunsch (NW)	Dynamic Programming	Bioinformatics	2048×2048 data points
HotSpot* (HS)	Structured Grid	Physics Simulation	500×500 data points
Back Propagation* (BP)	Unstructured Grid	Pattern Recognition	65536 input nodes
SRAD*	Structured Grid	Image Processing	512×512 data points
Leukocyte Tracking (LC)	Structured Grid	Medical Imaging	219×640 pixels/frame
Breadth-First Search* (BFS)	Graph Traversal	Graph Algorithms	1000000 nodes
Stream Cluster* (SC)	Dense Linear Algebra	Data Mining	65536 points, 256 dimensions
MUMmer (MUM)	Graph Traversal	Bioinformatics	50000 25-character queries
CFD Solver (CFD)	Unstructured Grid	Fluid Dynamics	97k elements
LU Decomposition* (LUD)	Dense Linear Algebra	Linear Algebra	256×256 data points
Heart Wall Tracking (HW)	Structured Grid	Medical Imaging	609×590 pixels/frame

layout. The CFD solver is released with two versions: one with precomputed fluxes, and the other with redundant flux computations. CFD is an implementation of the work by Corrigan *et al.* [11].

Motivation for inclusion: The CFD implementation applies data layout optimizations to reduce uncoalesced memory accesses to GPU memory. It also provides both single-precision and double-precision floating point implementations for the GPU, which allows users to analyze the trade-off between performance and computational precision. Also, computational fluid dynamics is widely regarded as a very important scientific workload.

MUMmerGPU (MUMmer): MUMmerGPU, developed by Schatz *et al.* [28], is an high-throughput, parallel, pairwise, local-sequence alignment program. It uses the GPU to simultaneously align multiple query sequences against a single reference sequence stored as a suffix tree encoded with 2D textures. The tree of the reference sequence is constructed on the CPU using Ukkonen's Algorithm [33] and transferred to the GPU along with the query sequences. The query sequences are then transfered to the GPU, and are aligned with the tree on the GPU.

Motivation for inclusion: The working set and code size of MUMmer is significantly larger than other benchmarks, stressing memory systems. Additionally, the suffix tree implementation presents the challenge of mapping Mummer's data structures to a GPU computational model while utilizing efficient data layouts.

TABLE II
GPGPU-SIM CONFIGURATIONS.

Parameter	Value	Parameter	Value
Clock Frequency	2 GHz	No. of CTAs/Core	8
No. of SMs	28	Number of Registers/Core	16384
Warp Size	32	Shared Memory/Core	32 kB
SIMD pipeline width	32	Shared Memory Bank Conflict	True
No. of Threads/Core	1024	No. of Memory Channels	8

III. CHARACTERIZATION OF RODINIA

In this section, we characterize Rodinia's applications in terms of instructions per cycle (IPC), memory instruction mix, and warp divergence. Our analysis shows that the Rodinia applications demonstrate good diversity, and the addition of new benchmarks enriches the application coverage of the suite beyond that of the previous Rodinia release [8]. We also use Rodinia to benchmark the NVIDIA GeForce GTX480 GPU (Fermi) targeting each of L1 and shared memory as preferred configurations.

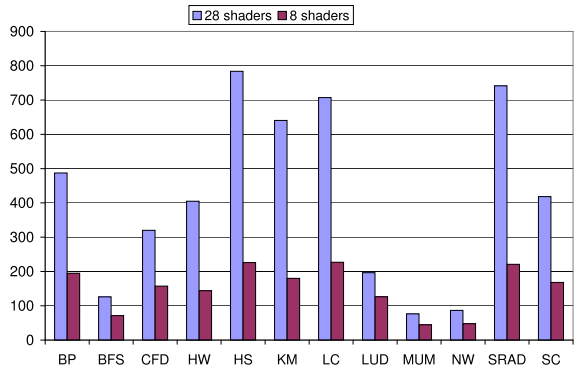


Fig. 1. IPCs are measured over 8-shader and 28-shader configurations.

A. Experiment Setup

To measure the execution characteristics of the Rodinia GPU benchmarks, we use GPGPU-Sim [2] from the University of British Columbia. GPGPU-Sim provides a detailed simulation model of a contemporary GPU capable of running CUDA and OpenCL workloads. Table II shows the parameters we used to configure the simulator.

Our GPGPU-Sim simulations did not use an L2 cache. Table I lists input details for the Rodinia applications we used in the simulations. In the Fermi benchmarking experiments, we use an NVIDIA GeForce GTX480 with 15 streaming multiprocessors (SMs) with a total of 480 1.4 GHz streaming processors (SPs), and a 768 kB L2. Each SM has a 64 kB, configurable, on-chip memory that can be configured as 48 kB shared + 16 kB L1 or as 16 kB shared + 48 kB L1. We use NVIDIA CUDA 2.2 for the GPGPU-Sim simulations (the simulator currently supports up to CUDA 2.3); for the GTX 480 experiments, we use CUDA version 3.0.

B. GPU Benchmark Results

Figure 1 shows the IPCs of each of the Rodinia benchmarks measured with 28-shader—the default configuration provided

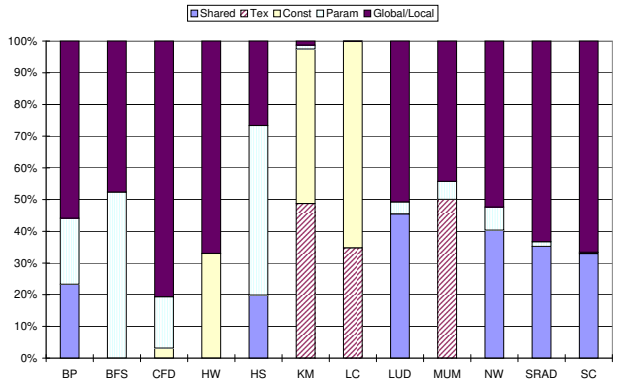


Fig. 2. Memory operation breakdown in terms of shared, texture, constant, parameter, and local or global memory instructions. "Param" memory refers to parameters passed through the GPU kernel call, which we always treat as cache hits [2].

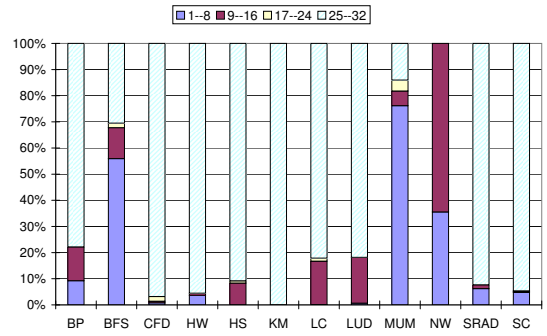


Fig. 3. Warp occupancies show the numbers of active threads in an issued warp over the entire runtime of the benchmark [2].

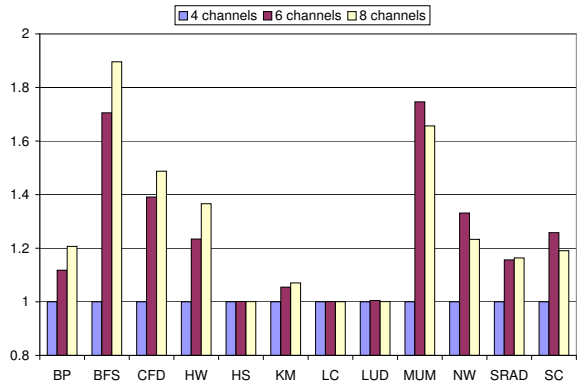


Fig. 4. Memory bandwidth improvements while increasing the number of memory channels. All data is normalized to the 4-channel case.

by the GPGPU-Sim package [2] with a SIMD width of 32—and 8-shader configurations. The IPCs with the 28-shader configuration range from less than 100 in *MUMmer* and *Needleman-Wunsch*) to more than 700 in *SRAD*, *HotSpot*, and *Leukocyte*. The highest IPCs are usually due to massive parallelism, better usage of memory locality, and good algorithmic optimization [6], [8], [24], [31]. Low IPC can be attributed to any of myriad faults: there is limited parallelism per iteration in *Needleman-Wunsch* due to the dependencies of processing data elements in a diagonal strip manner [9]; the overhead of the GPU's global memory accesses dominates *Breadth-First Search*; and some applications present many divergent branches. The benchmarks show high scalability across 8 and 28 shaders, except for those like *MUMmer* and *Breadth-First Search*, which are limited by the global memory access bandwidth, and like *LUD* with significant row and column dependencies.

Many Rodinia benchmarks take advantage of the GPU's specialized memory spaces by localizing data access patterns and inter-thread communication within thread blocks to take advantage of the SM's per-block shared memory. For read-only data structures, binding to cached constant or texture memory to reap the benefits of caching can provide significant performance improvements. Figure 2 shows a breakdown of different types of memory accesses. Applications such as *Back Propagation*, *HotSpot*, *Needleman-Wunsch* and *StreamCluster* make extensive use of shared memory. The performances of *Kmeans*, *Leukocyte* and *MUMmer* are improved by taking advantage of texture memory. Differing from *Kmeans* and *Leukocyte*, *Heartwall* uses constant memory to store large numbers of parameters which cannot be readily fit into shared memory.

Figure 3 shows warp occupancies [2]—the average number of active threads over all issued warps—over the entire runtime of the benchmarks. In a SIMT model [25], the cores will achieve the best performance when the threads within a SIMT group follow the same execution path. For example, because it must determine whether or not neighboring nodes have been visited, *Breadth-First Search* contains many control flow operations; hence the high number of low occupancy warps. *SRAD* does not have much control flow; what of it there is deals with the loading and processing data elements lying on the boundaries between data blocks. *Heartwall* must determine the specific operations to execute on the various regions of the image, but this requires a relatively small portion of the calculation, and the rest of the computation executes with little control flow.

For applications other than *Breadth-First Search*, *SRAD*, and *Heartwall*, unfilled warps are not due to branch divergence. Only some of the threads in *Back Propagation* are active, due to the parallel reduction; assuming a 16-element sum reduction, the number of active threads during the four iterations are 8, 4, 2 and 1. A similar situation occurs in *Needleman-Wunsch*, where, in each thread-block, the number of active threads is less than 16. *MUMmer*, in particular, experiences severe performance penalties because more than 60% of its warps have less than 5 active threads [2].

Figure 4 shows the bandwidth improvement as we increase the number of memory channels from 4 to 8. The benchmarks which benefit most from this change include *Breadth-First Search*, *CFD* and *MUMmer*. *LUD* and *HotSpot*, which take advantage of shared memory locality, benefit less from increased memory channels. For *Kmeans* and *Leukocyte*, little

improvement occurs with additional channels because we bind their main data structures to texture memory and make use of use constant memory.

C. Incrementally Optimized Versions

One important distinguishing characteristic of Rodinia is its support for multiple versions of individual benchmarks. These *incremental versions* are useful tools for architects and compiler developers because they allow analysis of the impact of hardware and software design choices on problems that are fundamentally the same but differ in certain specifics. Incremental versions can be used by programmers and compiler developers as "road maps" for similar problems, to aid them in getting from unoptimized to optimized or to evaluate their own optimizations.

We are preparing to release incremental code versions of *Leukocyte*, *LUD*, *Needleman-Wunsch* and *SRAD*. Table III shows sample characteristics of two different versions of *SRAD* and *Leukocyte*. We apply more shared memory optimization on the second version of SRAD, thus increasing the IPC from 404 to 748. Similarly, the performance of Leukocyte version 2 is improved by reducing the percentage of long latency global memory accesses through the use of persistent thread blocks. Boyer *et al.* provide a detailed study on the optimization of *Leukocyte* [6].

TABLE III
INCREMENTALLY OPTIMIZED VERSIONS OF SRAD AND LEUKOCYTE.

Benchmarks	Statistics	
SRAD	Version 1	IPC: 404, BW Utilization: 26% Shared: 9.7%, Global: 49.3% (Mem. inst. mix)
	Version 2	IPC: 748, BW Utilization: 34% Shared: 28.9%, Global: 51.9%
Leukocyte	Version 1	IPC: 656, BW Utilization: 8% Const: 54.1%, Tex: 22.7%, Global: 7.7%
	Version 2	IPC: 707, BW utilization: 3% Const: 65.1%, Tex: 34.7%, Global: 0.0%

D. Fermi Evaluation

Unlike the earlier G80 and Tesla products, NVIDIA's Fermi architecture includes traditional L1 and L2 caches. Each SM has 64 kB of on-chip memory that can be configured as 48 kB of shared memory and 16 kB of L1 (*shared bias*), the default configuration) or as 16 kB of shared memory and 48 kB of L1 (*L1 bias*. CUDA provides a new API function, `cudaFuncSetCacheConfig()`, to select the desired configuration [13]. A unified L2 cache handles all memory requests for data loads and stores, as well as all texture fetches.

Figure 5 shows the results obtained measuring the performance of the Rodinia CUDA implementations on an NVIDIA GeForce GTX480 GPU with each memory configuration. We compare to the results on a GTX280 GPU with 240 1.3 GHz SPs and 1 GB of device memory. All the measurements are kernel execution times normalized to the GTX280. Excepting *LUD* and *Leukocyte*, the total workload size of all benchmarks is larger than the aggregate L1 capacity.

The performances of *MUMmer* and *BFS*, which have large numbers of global memory accesses, improve by 11.6% and 16.7% respectively after switching the configuration from shared bias to L1 bias. Many Rodinia applications, including *SRAD*, *Needleman-Wunsch* and *Leukocyte*, which are designed to utilize shared memory well, expectedly prefer the shared bias setting. *LU Decomposition* and *StreamCluster* show very little performance variation between the two configurations.

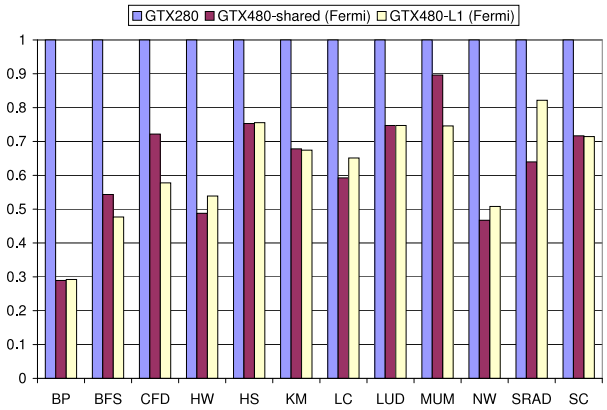


Fig. 5. Normalized kernel execution time of the GPU implementations on a GTX 280 and a GTX 480 (Fermi). Two configurations (L1 and shared bias) are used for the GTX 480 measurements.

E. GPU Sensitivity Study

To help architects achieve a desirable design, it is necessary to identify key architectural parameters and understand how benchmarks respond to changes in those parameters. Yi *et al.* [36] proposed using the Plackett-Burman (PB) design approach to determine the effect of a parameter on a processor's performance. Given n architectural parameters, PB requires only approximately $2n$ simulations instead of the 2^n required for brute-force full coverage. The downside of the PB methodology is its inability to effectively quantify the effects of all the interactions; however, Yi *et al.* show that if an interaction between parameters was significant, it was significant only because each of its constituent parameters was individually significant.

We use the PB approach to evaluate a subset of GPU architectural parameters that are important to performance. The parameters we choose include core clock rate (1.2 GHz–1.5 GHz), SIMD width (16–32), shared memory size (16 kB–32 kB), bank conflict (on or off), register file size (16384–32768 registers), number of threads (1024–2048), memory clock (800 MHz–1 GHz), number of memory channels (4–8), and DRAM bus width (4–8 bytes). We use GPGPU-sim to configure these parameters and evaluate the resulting changes of total execution cycles with the 11-column PB matrix.

SIMD width and the number of memory channels have the largest impacts on benchmark performance, often demonstrating more than an order of magnitude greater effect than other parameters; however, different applications show different characteristics. For instance, *SRAD* makes extensive uses of the shared memory; for this application, the configurations of shared memory and the number of memory channels have similar impact on performance. Similarly, shared memory bank

conflict, SIMD-width, and memory bandwidth demonstrate similar influence on performance for *Needleman Wunsch*. This is attributable to the fact that the current implementation processes diagonal strips on a 16×16 shared memory block, which leads to copious bank conflict and suggests that we have space for further optimization. The applications, including *Leukocyte* and *HotSpot*, which have better on-chip data locality or utilize texture memory units, are impacted only modestly by changes to the memory interfaces.

Note that some complex interactions between parameters cannot be captured by Plackett-Burman. For example, the number of thread blocks that can be issued is often limited by another resource limitation of the GPU (the number of registers or threads, or the shared memory size). It is possible that once limited by one constraint, providing extra resources for structures does not lead to any performance improvement; this is a question we plan to explore in future work.

IV. Rodinia and Parsec

This section answers several important questions: 1) How do workloads, like Rodinia, which are designed for heterogeneous platforms differ from those of other suites designed for multicore CPUs, like Parsec? 2) How well do the chosen applications span the workload space? 3) What aspects of Parsec and Rodinia are differentiating? We hope that examination of these questions may facilitate the improvement of workload construction for multicore CPU and accelerator performance analysis.

How to perform fair comparisons between accelerator and CPU workloads running on different architectures is an open research question, and one which we cannot adequately address in this paper. Among the difficulties in heterogeneous, parallel benchmarking are the questions of 1) algorithm choice: How alike are the underlying algorithms of two different implementations? 2) optimization: What does it mean to compare the quantity and quality of optimization across heterogeneous platforms? 3) effort: If performance is not the sole concern, the next item on the list is probably cost or programmer effort. How difficult is an application to implement [8]–[10]?

The Rodinia OpenMP and CUDA implementations are developed congruously, using same algorithms with similar levels of optimization; we reserve formal evaluation of their similarity for future work. We use the Rodinia OpenMP implementations to compare with the Parsec benchmarks in this study. We apply principal component analysis (PCA) to identify distinctions and also to characterize the workloads in terms of cache behavior, working set, and other, similar performance metrics.

A. Comparison of Rodinia and Parsec

Parsec, a benchmark suite jointly developed by Princeton University and Intel, has been gradually gaining popularity among users of multithreaded workloads. The suite includes some workloads from emerging application domains and uses some state-of-the-art software techniques. Bienia *et al.* [4] compare SPLASH-2 and Parsec to determine the extent of feature overlap, and conclude that the workloads have significant differences. Many Parsec workloads have larger working sets than those in SPLASH-2, useful in the face of the scientific trend toward massive data growth. Other work compares the communication characteristics of SPLASH-2 and Parsec [34] and examines the behavior of Parsec on real hardware [3].

TABLE IV
Comparison between Parsec and Rodinia.

Features↓ \ Suite→	**Parsec**	**Rodinia**
Platform	CPU	CPU and GPU
Programming Model	Pthreads, OpenMP, and TBB	OpenMP and CUDA
Machine Model	Shared Memory	Shared Memory and Offloading
Application Domains	Scientific, Engineering, Finance, Multimedia	Scientific, Engineering, Data Mining
Application Count	3 Kernels and 9 Applications	6 Kernels and 6 Applications
Optimized for...	Multicore	Manycore and Accelerator
Incremental Versions	No	Yes
Memory Space	HW Cache	HW and SW Caches
Problem Sizes	Small–Large	Small–Large
Special SW Techniques	SW Pipelining	Ghost-zone and Persistent Thread Blocks
Synchronization	Barriers, Locks, and Conditions	Barriers

Table IV provides a high-level overview of the differing design focuses of Parsec and Rodinia, while Table V provides some more specific details on Parsec. In the previous sections, we discussed several aspects of Rodinia which distinguish it from other benchmark suites; here we provide some more discussion on the topic, specifically with respect to Parsec.

Parsec provides a rich set of features that support fine-grained parallelism (locks), languages (TBB, OpenMP, and Pthreads), and large code bases. Rodinia currently focuses only on OpenMP workloads for the CPU implementations. The use of fine-grained parallelism in Rodinia, even in CPU implementations, is restricted by our desire to maintain algorithmic congruence with the CUDA ports given the fact that CUDA supports only barrier synchronization within a thread block [22] and global synchronization at kernel exit or when using a global synchronization primitive. In the construction of the Rodinia benchmark suite, we also consider Parsec workloads. We include *StreamCluster* in Rodinia, but find that those benchmarks relying on task pipelining, like *Ferret*, do not port well unless each stage is also heavily parallelizable.

B. Methodology

To compare Rodinia and Parsec, we adopt the methodology and metrics of Bienia *et al.* [4] in their SPLASH-2 and Parsec comparison, so that the reported results are cross-comparable. The points of comparison include instruction mix (including ALU, branch, and memory instructions), working set (cache misses per memory reference), and sharing behavior (the fraction of cache lines shared, and the number of accesses to shared lines per memory reference). Our experiments use eight cache sizes, ranging from 128 kB to 16 MB, and measure the sharing and the working set behavior. We adopt a similar cache structure to that used by Bienia *et al.* as well, an 8-core processor with a single cache shared by all cores. The cache is 4-way associative with 64 byte lines. All programs are compiled with gcc 4.2.1 with OpenMP or Pthreads.

All data is obtained with Pin [23]. Pin is a dynamic, binary instrumentation tool that instruments an application that executes on Intel processors. It provides an infrastructure for

TABLE V
PARSEC APPLICATIONS AND SIM-LARGE INPUT SIZES. [3], [4]

Application	Application Domain	Problem Size	Description
Blackscholes	Financial Analysis, Algebra	65,536 options	Portfolio price calculation using Black-Scholes PDE
Bodytrack	Computer Vision	4 frames, 4,000 particles	Computer vision, tracks 3D pose of human body
Canneal	Engineering	400,000 elements	Synthetic chip design, routing
Dedup	Enterprise Storage	184 MB	Pipelined compression kernel
Facesim	Animation	1 frame, 372,126 tetrahedrons	Physics simulation, models a human face
Ferret	Similarity Search	256 queries, 34,973 images	Pipelined audio, image and video searches
Fluidanimate	Animation	5 frames, 300,000 particles	Physics simulation, animation of fluids
Freqmine	Data Mining	990,000 transactions	Data mining application
StreamCluster	Data Mining	16,384 points per block, 1 block	Kernel to solve the online clustering problem
Swaptions	Financial Analysis	64 swaptions, 20,000 simulations	Computes portfolio prices using Monte-Carlo simulation
Vips	Media Processing	1 image, 26,625,500 pixels	Image processing, image transformations
X264	Media Processing	128 frames, 640,360 pixels	H.264 video encoder

writing program analysis tools, called *Pin tools*. Instruction mix is obtained using the *mix-mt* tool provided with the Pin package. We developed our own Pin tool, based on the cache tool in Pin, with support for multithreaded workloads to collect cache behavior characteristics.

C. Principal Component Analysis and Measuring Similarity

Principal components analysis (PCA) is a statistical, data analysis technique that reduces a data set's dimensionality and removes correlation from the data set while controlling the amount of information lost. PCA computes n new variables, called principal components, which are linear combinations of n original variables, such that all principal components are uncorrelated. The first of the resulting orthogonal principal components exhibits the largest variance, followed by the second, followed by the third, and so on [15]. After performing PCA, we cluster to find equivalence classes of programs with similar characteristics. PCA has been widely applied for benchmark comparison [4], [15], [18], [26] in similar contexts; however, the question of how to perform more fair and accurate evaluation and comparison of benchmarks is an open one and beyond the scope of this paper.

To measure the similarity among benchmarks, we use classical hierarchical clustering analysis. Similar approaches have been used in other recent performance analysis work. Clusters are formed in such a way that data objects in the same cluster are very similar and data objects in different clusters are very distinct. We use the MATLAB [32] statistics toolbox to process the data for the collected characteristic values for all the benchmarks. The algorithm involves finding the similarity or dissimilarity between every pair of data objects in the data set using distance functions and grouping the objects into a binary, hierarchical cluster tree. Dendrograms are used to illustrate our results.

V. ANALYSIS

Here we present the results of our principal component analysis.

A. Hierarchical Clustering

Figure 6 shows the overlap of the two program collections. In the figure, the magnitude of the link between any two nodes (or clusters of nodes) quantifies the measure of dissimilarity

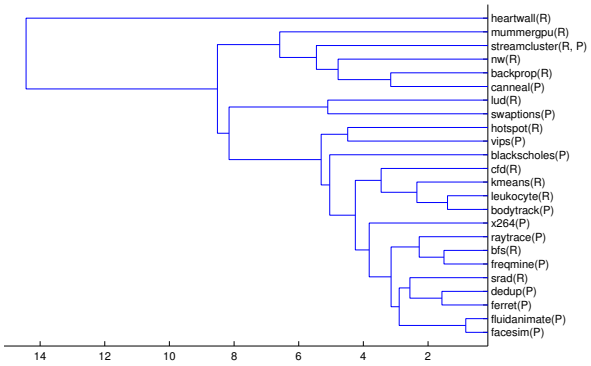


Fig. 6. A dendrogram showing the similarity between the Parsec (P) and Rodinia (R) workloads. The x axis represents the linkage distance in the PCA coverage space, which has no obvious physical analog.

between those nodes; thus, *Leukocyte* and *Bodytrack* are fairly similar, while *Heartwall* differs significantly from all other compared benchmarks; and *MUMmer* and *Swaptions*, while spatially close in the figure, are more dissimilar than *HotSpot* and *Facesim*. From this dendrogram, it is evident that the two benchmark suites cover similar application spaces, with most clusters containing both Rodinia and Parsec applications. Also note that the new applications added to Rodinia, namely *CFD*, *LUD*, *MUMmer*, and *Heartwall*, enrich the original application set, with the latter two significantly different from all others.

We also perform an analysis of some subsets of our characteristics: instruction mix, cache miss rate, and data sharing behavior. Instruction mix is an interesting metric because it represents a set of fundamental program characteristics and the utilization of various hardware components, while the cache miss rate characterizes data locality and reuse, and an application's sharing behavior is important to multithreading workloads and communications [4].

Figures 7, 8 and 9 show the instruction mix, working set and sharing behaviors of the programs as is similarly shown in the Parsec and SPASH-2 comparison [4]. In Figure 7, Parsec and Rodinia demonstrate disparate behavior, with *Breadth-First Search*, *Back Propagation*, and *HotSpot* from Rodinia, and *Raytrace*, *Ferret*, *Bodytrack*, and *StreamCluster* from Parsec

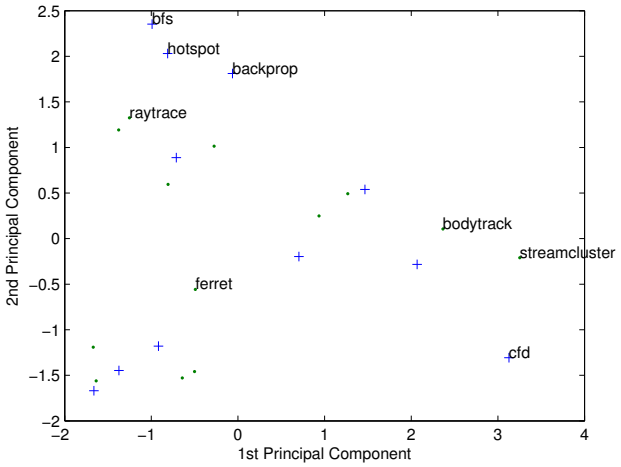


Fig. 7. The instruction mix plot using two PCA components for Parsec (dots) and Rodinia (crosses).

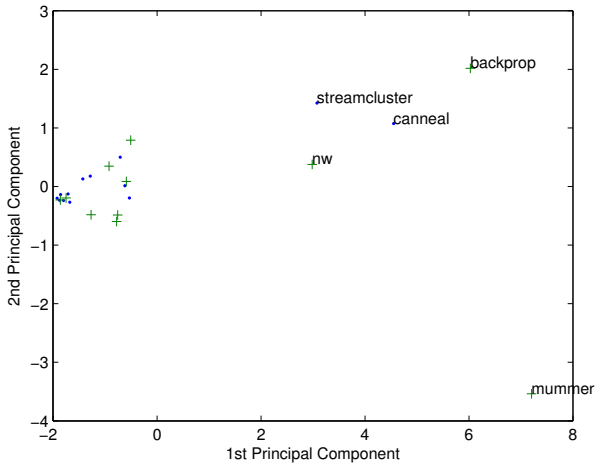


Fig. 8. The working set plot using two PCA components for Parsec (dots) and Rodinia (crosses).

tending to populate different areas in the space. In the working set plot of Figure 8, there are several Parsec and Rodinia benchmarks that are clear outliers from the main cluster; *MUMmer* is a significant outlier, which correlates with its high miss rates. The miss rates—given in cache misses per memory reference—of all the benchmarks under a 4 MB cache configuration are shown in Figure 10. Figure 9 shows similar behavior—data sharing, now, rather than working set size—with *Heartwall* significantly different from the rest. Looking back at Figure 6, *Heartwall* and *MUMmer* are the most disparate benchmarks in the suite; something which is backed up by this series of figures.

As is evident from these figures, Rodinia clearly provides a good workload mix for multicore CPUs. Additionally, Parsec and Rodinia demonstrate features that complement with each other, suggesting that researchers should consider both of them, possibly as well as other benchmark suites, to ensure a reasonable application coverage for their work.

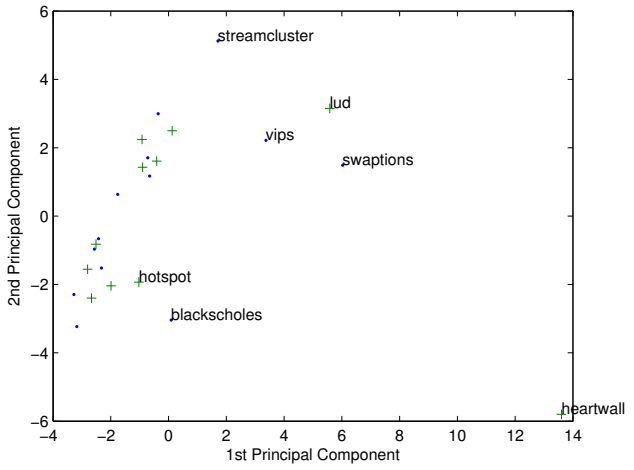


Fig. 9. The sharing plot using two PCA components for Parsec (dots) and Rodinia (crosses).

B. Clustering Discussion

How well is the application space covered by the two suites?

— Our clustering analysis shows that Parsec and Rodinia cover similar application spaces. This does not imply that using either or both of them is sufficient for research. Consider the blank regions in the PCA coverage spaces of Figures 7, 8, and 9; it is unclear whether these regions can be covered by other real-world workloads or benchmark suites. This implies that a thorough examination requires a comprehensive evaluation and comparison of all the current multithreaded benchmark suites, including SPLASH-2 and various domain-specific workloads, to establish a single set of workloads with sufficient coverage and little redundancy. Previous work performs such studies but only on single-threaded benchmarks [18], [26], while this is an open problem in heterogeneous environments.

The metrics evaluated in this work are important for multithreaded program behavior [4]. On the other hand, other potentially important metrics may indicate other crucial differences between benchmarks. It is an area of ongoing research to develop a set of metrics which are able to capture most behaviors of multithreaded workloads. Host *et al.* [15] propose a set of microarchitecture-independent workload characteristics to profile single-threaded applications, and which are also useful for performance prediction [16]. A set of metrics for multithreaded workloads are needed.

Can the Parsec workloads be effectively mapped to heterogeneous platforms?

— Our clustering results indicate that, except for a few

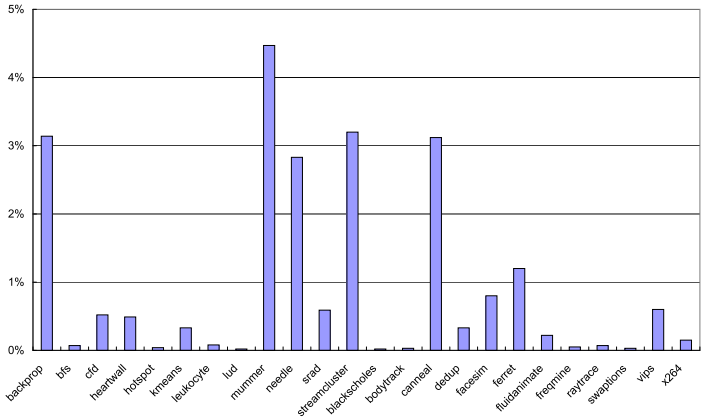


Fig. 10. The miss rates of Rodinia and Parsec [4] benchmarks under a 4 Mb cache configuration.

outliers, the heterogeneous workloads we developed for Rodinia are not fundamentally different from those of Parsec, developed for multicore CPUs. This, however, does not imply that since all the Rodinia benchmarks map well to the GPU platform, the same will be true of the Parsec benchmarks. We have found many challenges in the task of porting traditional, multithreaded, CPU workloads onto heterogeneous platforms. Some issues which make this less than straightforward:

- **Library Modules**: Application development depends upon libraries and reuse for productivity and maintainability. This poses a potentially large challenge in porting CPU applications to accelerators. Though it is possible to implement each library module on the GPU, for example, the cost of maintaining modularity is the possibly resultant overhead of GPU kernel call invocation and memory transfer between the CPU and the GPU. To achieve better performance for GPU applications, optimization sometimes requires cross-function algorithmic reorganization, or the division of a logical function into multiple kernels.
- **Synchronization**: Many Parsec applications heavily rely on fine-grained synchronization primitives, such as mutexes [5]. For some applications, like *StreamCluster*, it is relatively easy to reorganize for the GPU, while for others, it is non-trivial; especially in those applications using the software pipelining model, including *Dedup* and *Ferret*, which require significant algorithmic reorganization. The difficulty in supporting these primitives is directly attributable to the GPU's limited synchronization capabilities. On the GPU, synchronization within a thread block is provided, and global synchronization is achieved via a barrier primitive. The latest CUDA versions also provide a primitive for an on-chip, global memory fence which, unfortunately, requires restructuring of applications such that thread blocks are persistent during the entire program execution. Locks across thread blocks are non-trivial to implement, and performance benefits are not guaranteed.

Are existing application classification taxonomies sufficient to differentiate application characteristics?

Several approaches have been proposed for classifying applications based on their memory access and execution patterns, including the Berkeley *Dwarves* [1] and Intel's *Recognition, Mining and Synthesis* (RMS) [21]. Rodinia and Parsec were designed with the Dwarves and RMS as guidelines, respectively. Although these taxonomies are defined at a high level of abstraction to provide useful guiding principals and to allow users to effectively reason about program behavior, our work, often with multiple instances of a single Dwarf, suggests that the Dwarf taxonomy alone may not be sufficient to ensure adequate diversity, and that some important behaviors may not be captured by the Dwarves.

As shown in Figure 6, for *Structured Grid* applications, stencil-type workloads, such as *SRAD* and *Fluidanimate*, are quite similar. However, applications such as *HotSpot*, *Leukocyte*, and *Heartwall* are located in different clusters, with *Heartwall* significantly different from the others. *Back Propagation* and *CFD* are both from the *Unstructured Grid* Dwarf and show significant differences. The *Graph Traversal* applications, *MUMmer* and *Breadth-First Search*, are also very dissimilar.

Even applications from the same application domain are quite different; for example, the two fluid dynamics applications, Parsec's *Fluidanimate* and Rodinia's *CFD* differ more than *Fluidanimate* and *Facesim*, the latter members of different Dwarves. Also, two data mining benchmarks, *Kmeans* and *StreamCluster*, both of which rely on distance-based clustering, lie far apart in the binary clustering tree.

C. Instruction and Data Footprints

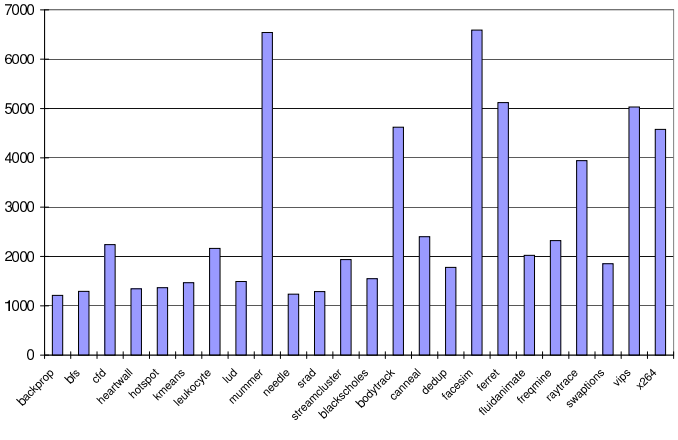


Fig. 11. The numbers of 64-byte instruction blocks touched during the program execution.

Figures 11 and 12 illustrate the instruction and data footprints of Parsec and Rodinia. The figures show the number of 64-byte instruction blocks and 4 kB data blocks touched during the entire program execution [18]. Figure 12 shows that both Parsec and Rodinia use large working sets, but, with the exception of *MUMmer*, Parsec applications tend to have larger instruction footprints, or code sizes, than Rodinia workloads.

There is a related, open question in workload characterization, that of the difference between "big" applications and "small" ones, or, in other words, between applications and kernels. Better understanding this issue requires finding the "building blocks" of the applications and a method to correlate applications with constituent kernels. Carrington *et al.* [7] did this in the HPC domain, but some more sophisticated approaches are needed for higher prediction accuracy.

VI. Related Work

The Parsec benchmark suite [5] includes emerging applications from finance, multimedia, and data mining. Parsec benchmarks utilize relatively large working sets and are developed with state-of-the-art software techniques such as software pipelining. Some earlier benchmark suites include SPLASH-2 [35] and SPEC OMP2001 [29], consisting of general-purpose workloads focusing on science, engineering, and graphics. BioParallel [17], ALPBench [20], and MineBench [27] target specific application domains.

Parboil [30] and SHOC [12] are two efforts to benchmark GPUs, but the former does not provide any diversity analysis and the latter targets systems with multiple GPU nodes. Bakhoda *et al.* developed GPGPU-Sim [2] and use it to analyze various CUDA programs. Hong *et al.* developed an analytical model to predict GPU performance [14] and proposed metrics to represent degree of warp-level parallelism. Kerr *et al.* proposed a set of metrics for GPU workloads [19] and use these metrics to analyze the behavior of GPU programs. They also use these metrics combined with PCA and regression modeling to predict GPU performance. Rodinia is distinct from these works primarily in that it is designed to provide implementations with diverse parallel execution patterns, optimizations, and software mappings, in addition to its ability to compare platforms, a crucial capability for tackling the design challenges of future parallel and heterogeneous systems.

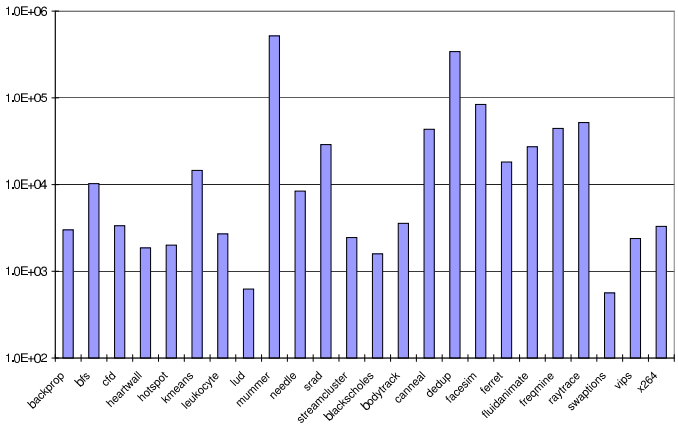


Fig. 12. The number of 4 kB data blocks touched during the program execution.

VII. Conclusions and Future Work

In this paper, we performed a detailed characterization of Rodinia, designed to let researchers better understand this collection of benchmarks and to show that the new applications—*Heartwall*, *CFD*, *LUD*, and *Mummer*—enrich the diversity of Rodinia.

We also compared Rodinia with Parsec. Some important differences we observe show the importance of measuring how well existing suites span the design space and the importance of using applications from different suites together.

Our experimental results show that Rodinia applications demonstrate a good mixture of diversity in terms of both basic program characteristics and how they stress accelerator architectures. Our applications demonstrate multiple degrees of data parallelism, branch divergence, and sensitivity to memory constraints.

Directions for future work include:

- conducting more detailed characterizations on the Rodinia GPU implementations, such as branch divergence sensitivity, data sharing among threads, and the impact of hardware thread scheduling mechanisms.
- identifying a set of metrics to quantify the extent to which the same algorithm exhibits different properties when implemented on different architectures.
- performing an application-space coverage study of existing multithreaded workloads.
- correlating program characteristics across the CPU and the GPU, as well as across big applications and small kernels.
- adding new features to the suite, including support for OpenCL and simultaneous kernel execution.

Acknowledgements

This work is supported by NSF grant nos. IIS-0612049, CNS-0916908 and CNS-0615277, a grant from the SRC under task no. 1607, and grants from NVIDIA Research and NEC labs. We would like to acknowledge George Mason University who allowed us to use their CFD application, and the University of Maryland who contributed their MUMmerGPU implementation.

References

[1] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] A. Bakhoda, G. L. Yuan, W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009.

[3] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct 2009.

[4] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Sep 2008.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct 2008.

[6] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating Leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, May 2009.

[7] L. Carrington, M. Laurenzano, A. Snavely, R. Campbell, and L. Davis. How well can simple metrics represent the performance of HPC applications? In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Nov 2005.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, Lee S-H, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct 2009.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[10] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute intensive applications with GPUs and FPGAs. In *Proceedings of the 6th IEEE Symposium on Application Specific Processors*, June 2008.

[11] Andrew Corrigan, Fernando Camelli, Rainald Löhner, and John Wallin. Running unstructured grid cfd solvers on modern graphics hardware. In *19th AIAA Computational Fluid Dynamics Conference*, June 2009.

[12] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable HeterOgeneous computing (SHOC) benchmark suite. In *Proceedings of Third Workshop on General-Purpose Computation on Graphics Processing Units*, Mar 2010.

[13] NVIDIA CUDA Programming Guide. Web resource. http://developer.nvidia.com/object/gpucomputing.html.

[14] S.P Hong and H.S Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th International Symposium on Computer Architecture*, June 2009.

[15] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[16] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2006.

[17] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb 2006.

[18] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55(6):769–782, 2006.

[19] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of PTX kernels. In *Proceedings of the 2009 International Symposium on Workload Characterization*, Oct 2009.

[20] M. Li, R. Sasanka, S. V. Adve, Y. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, Oct 2005.

[21] B. Liang and P. Dubey. Recognition, mining and synthesis moves computers to the era of Tera. *Technology@Intel Magazine*, Feb 2005.

[22] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[23] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[24] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd Annual ACM International Conference on Supercomputing*, June 2009.

[25] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[26] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th International Symposium on Computer Architecture*, June 2007.

[27] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical Report CUCIS-2005-08-01, Department of Electrical and Computer Engineering, Northwestern University, Aug 2005.

[28] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.

[29] The Standard Performance Evaluation Corporation (SPEC). Web resource. http://www.spec.org.

[30] Parboil Benchmark suite. Web resource. http://impact.crhc.illinois.edu/parboil.php.

[31] L. G. Szafaryn, K. Skadron, and J. J. Saucerman. Experiences accelerating MATLAB systems biology applications. In *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits*, June 2009.

[32] MATLAB Statistics Toolbox. Web resource. http://www.mathworks.com.

[33] E. Ukkonen. On-line construction of suffix trees, 1995.

[34] N. B. Williams, C. Fensch, and S. Moore. A communication characterization of SPLASH-2 and PARSEC. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct 2009.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[36] J. J. Yi, D. J. Lilja, and D. M. Hawkins. A statistically rigorous approach for improving simulation methodology. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, Feb 2002.

Data Handling Inefficiencies between CUDA, 3D Rendering, and System Memory

Brian Gordon, Sohum Sohoni, Damon Chandler
Electrical and Computer Engineering Department, Oklahoma State University
(brian.gordon,sohum.sohoni,damon.chandler)@okstate.edu

Abstract – **While GPGPU programming offers faster computation of highly parallelized code, the memory bandwidth between the system and the GPU can create a bottleneck that reduces the potential gains. CUDA is a prominent GPGPU API which can transfer data to and from system code, and which can also access data used by 3D rendering APIs. In an application that relies on both GPU programming APIs to accelerate 3D modeling and an easily parallelized algorithm, the hidden inefficiencies of nVidia's data handling with CUDA become apparent. First, CUDA uses the CPU's store units to copy data between the graphics card and system memory instead of using a more efficient method like DMA. Second, data exchanged between the two GPU-based APIs travels through the main processor instead of staying on the GPU. As a result, a non-GPGPU implementation of a program runs faster than the same program using GPGPU.**

I. Introduction

GPGPU programming (General-Purpose computation on Graphics Processing Units) is becoming a popular method of accelerating parallel workloads and achieving substantial speedup. As GPGPU gains popularity, new programming APIs are written to provide a layer of abstraction to programmers to allow their code to run on any graphics card. Emerging APIs like OpenCL [1] and DirectCompute [2] allow programmers to write code for any graphics card, while existing APIs like nVidia's CUDA [3] allow code to execute only on nVidia's GPUs. There have already been several successful applications of GPGPU in the areas of image processing, encryption, and database applications (see, e.g., [4],[5],[6]).

Interestingly, there was similar development on the first GPU acceleration APIs, 3D rendering. Early 3D rendering was handled by the CPU, but the calculations moved to highly parallelized hardware, and APIs like Direct3D and OpenGL abstract the underlying hardware. Modern nVidia consumer graphic cards support both the graphics acceleration and general-purpose computing within the same application with the possibility of sharing data between the two APIs. The benefits of using both APIs are twofold: (1) Either method can develop data for the other to use, utilizing the specialization of each API. (2) Applications that depend on both types of data-processing can benefit from the specialized hardware.

In this paper, we describe an application that utilizes both, the 3D rendering specialization of graphics cards, and the massively parallel computational power of the GPU, to accelerate an alternative video compression method called Model-Based Coding (MBC) [7]. The MBC encoding process (explained in detail in Section III) requires image processing of the 2D image of a 3D model. These requirements make MBC a natural candidate for the graphics processor, since both the 3D rendering and GPGPU capabilities of the graphics card can be utilized. This paper analyzes how the application and system performance is affected by utilizing two GPGPU programming methods in the MBC encoder, and it compares and contrasts the advantages and disadvantages of each method.

II. Related Work

Numerous papers discuss the advantages of GPGPU programming. 3D rendering APIs have been used for arbitrary calculations using the programmable stages of the 3D rendering pipeline. Most commonly, image processing [8-10] can benefit greatly using pixel shaders due to the specialized architecture of a GPU. However, utilizing a 3D rendering API for general-purpose computations requires adapting a specialized solution for something it was not designed to do. When nVidia added the ability to run general-purpose code on their hardware, many researchers adapted highly parallelizable algorithms to run on CUDA [4-6, 11]. However, these papers discuss only their implementation on GPU hardware and many touch only on algorithmic optimizations to maximize performance for the architecture.

Only recently have researchers begun to explore the limitations of GPGPU programming. Amorim *et al.* [12] look at a single problem, Jacobian iterations, and test its performance using both OpenGL pixel shaders and CUDA. They also test the performance of utilizing the different internal formats and specialized memories of each API.

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

Kothapalli *et al.* [13] look at modeling CUDA kernel performance based on its instruction types, memory accesses, and execution parameters. Using their models, the authors were able to estimate kernel runtime in the programs they profiled across a range of input data sizes. Suda *et al.* [14] compare CUDA's architecture and execution model to traditional cluster-based parallel computers. They also look at the potential shortcomings of CUDA and proposed a pipelining method kernel scheduling to hide the unavoidably large latencies of coping data to CUDA. Coutinho *et al.* [15] are developing methods of profiling CUDA programs to gauge performance bottleneck. Using their profiler, the researchers were able to identify bottlenecks in two CUDA programs by analyzing the time spent performing different tasks using the native CUDA PTX code. Researchers are now beginning to explore GPGPU implementations to identify the limitations and the types of applications that can truly benefit from GPGPU programming.

III. Model Based Video Coding

Traditional video compression algorithms such as MPEG-2/4 and H.264/AVC [16], [17] treat video frames as signals described in a statistical framework. Compression is achieved by applying a discrete cosine transform (DCT) to blocks within each frame (or differential frame), then quantizing the DCT coefficients, and then entropy encoding the quantized data. To fit within strict bandwidth constraints, such algorithms increase the amount of irreversible (lossy) compression, but this process can result in severely degraded video quality (e.g., severe blocking artifacts).

In Model Based Coding (MBC), rather than compressing frames of video, the system analyzes each frame to compute a small set of 3D model parameters. These parameters are sufficient to describe the objects-of-interest to a given level of detail. Only these parameters and not DCT coefficients are stored and/or transmitted. During decoding, a computer-rendering system uses these model parameters to create a rendition of the original scene.

MBC has the potential to radically improve video-conferencing and face-to-face video applications such as distance education. This is not only due to its extremely low bandwidth requirement, but also due to its valuable features not found in traditional video compression such as the ability to use interchangeable models, unlimited scalability with screen size (irrespective of the available communication bandwidth), and the ability to adapt to different external and user-chosen inputs such as ambient lighting and model detail. Although these benefits are known to the video-coding community, MBC is not currently popular due to the computational complexity required to fit a 3D model to a 2D video.

Specifically, transforming a captured video sequence into a model-based representation requires analysis of both spatial and temporal properties of the source video. This model-based analysis attempts to fit a 3D model to the source video by using a combination of computer vision, geometric modeling, and optimizations.

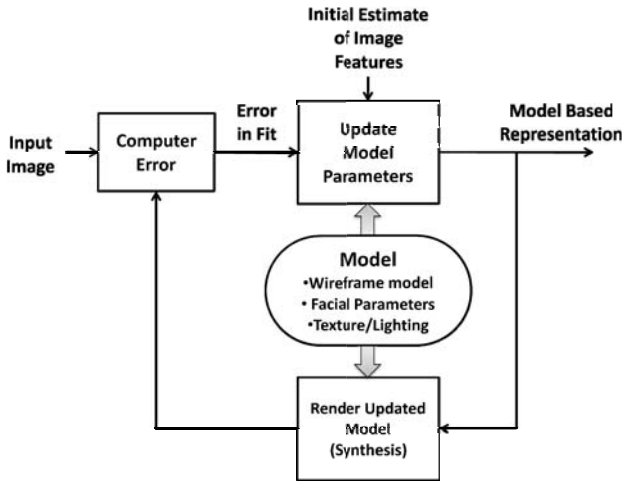


Figure 1. Typical steps involved in the analysis stage of a model-based coder (analysis by synthesis).

The primary bottleneck in model-based analysis stems from the optimization procedure employed during model-fitting, making MBC at least an order of magnitude slower than the DCT/motion-compensation-based analysis used in MPEG 2/4 and H.264/AVC [18]. Figure 1 depicts the steps employed in model-based analysis of facial video. The algorithm depicted uses a technique called *analysis-by-synthesis*, which consists of the following steps:

1. Estimate model parameters (e.g., facial expression, pose) using feature analysis and previous frames
2. Synthesize the face using estimated parameters;
3. Calculate the error between the synthesized face and actual face, e.g., using peak signal-to-noise ratio (PSNR).
4. Estimate the new pose and facial expression based on the error
5. Iterate until the algorithm converges on a minimum error (best estimate).

IV. GPGPU Implementations

A. Initial Analysis

Research in MBC has almost exclusively focused on encoding of facial video sequences (e.g., [7, 19-20]). MBC of faces has also been standardized in the MPEG-4 Facial Animation (FA) specification [21]. Accordingly, our analyses described in the following sections have been performed on an MBC implementation which follows the MPEG-4 FA standard. The analysis-by-synthesis algorithm

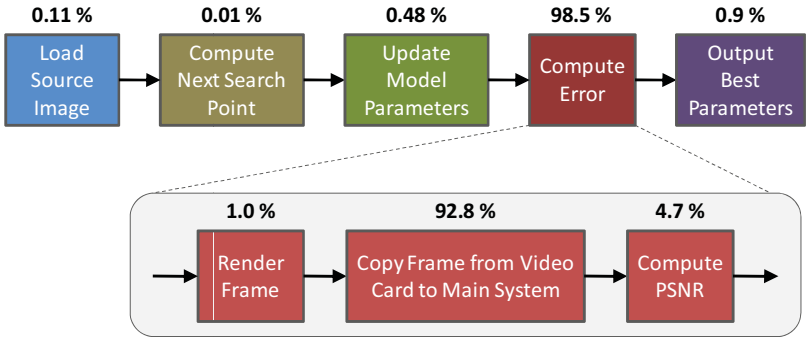


Figure 2. Profiling analysis of the model-fitting search procedure showing average computation time for each stage. It is clear that the error computation phase is the bottleneck.

also uses the MPEG-4 Facial Animation specification for the model's facial animation parameters (FAPs) to store the 3D head's model parameters. The encoder uses the GPU and Direct3D to render the model and uses the CPU to perform all the other steps.

Initial analysis of this program showed that it could not encode 30 frames-per-second video in real-time. Real-time encoding is an important requirement if MBC is intended for video conferencing. Analysis of the time spent within the functional blocks of the program showed that the vast majority of the time was spent in the error calculation. Upon closer inspection, the bottleneck was identified as the transfer of pixel data from the graphics card. Figure 2 shows the initial results.

While we could reduce the number of FAPs the encoder analyzes or reduce the image size to reduce the runtime, the resulting encoding quality would be drastically reduced, removing the benefits of MBC. Instead, we looked to mitigate the data transfer bottleneck by moving part of the error calculation to the graphics card and utilizing its general-purpose GPU capabilities. A diagram of the differences between each process is shown in Figure 3. With the CPU-based method, the reference image represented by the *R* head resides in CPU-accessible memory. The guessed image, the *G* head, must be copied from GPU-accessible memory to CPU memory to calculate the mean-square error (MSE) denoted in the figure by the symbol *e*. With the GPU-based MSE, the reference head is copied to GPU memory beforehand and the MSE is calculated on the GPU. The error value is then copied back to CPU memory to continue the encoding processes.

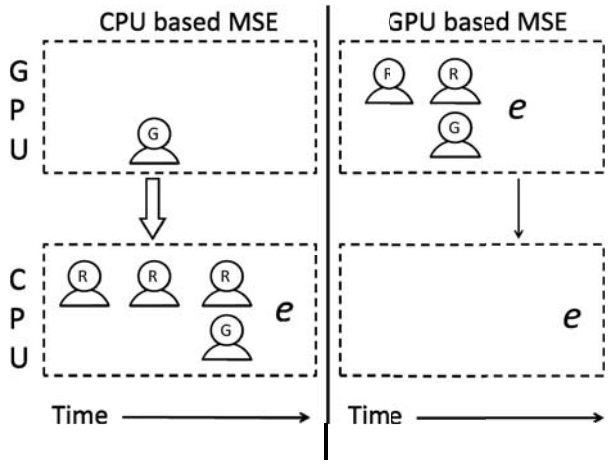


Figure 3. Moving the MSE calculation to the GPU reduced the amount of data transferred from the GPU from hundreds of kilobytes of pixel data to the four byte error result.

Moving the error calculation to the GPU would solve two problems with a CPU-based approach to the error calculation. First, the calculation of the MSE,

$$e = \frac{1}{N}\sum_{i=1}^{N}(\hat{x}_i - x_i)^2, \qquad (1)$$

an intermediate value used to calculate PSNR, is a highly data-parallel calculation. The subtraction and squaring parts of the MSE equation in particular computes the squared difference between two values $\hat{x}_i$ and x_i from the guessed image and the reference image respectively at each of the N data values. Since the subtraction and squaring uses independent data to calculate each result, this part of the calculation will benefit greatly from parallelization utilizing the high number of processors on a GPU to reduce the calculation time. Secondly, performing the MSE calculation on the GPU will keep the large amount of pixel data on the graphics card, removing the need to transfer all that data to system memory. Instead, only the result will need to be returned from the GPU.

B. Direct3D Pixel Shaders

One of the first approaches used to implement GPGPU programming was to utilize the programmable shader engines in 3D rendering APIs. In order to increase the realism of 3D rendering, previously fixed functions of the rendering pipeline were made programmable. Graphics card

manufactures opened up the vertex and, more importantly, pixel processing engines to support user-programmable code for new rendering techniques. User-programmable code allows developers to create more realistic mathematic models for calculating the final colors of the pixels. Pixel-shader programs are designed to calculate a final value for a single pixel [22]. Early attempts at GPGPU programming exploited this user-programmable feature to accelerate easily parallelized code. Input data was given to the graphics card as matrices masquerading as textures, shader programs were written to perform the necessary calculations for a single output value, and the resulting data was read back as the rendered image. Implementing GPGPU in a 3D rendering API can be difficult to program since it requires an understanding of the API to 'trick' the graphics card into properly processing the data.

Since our code already uses a 3D rendering API to generate one half of the input data for the error calculation, a second, separate rendering pass can perform the error calculation. To turn the rendered image of the head into an input for the error calculation, the output storage location of the first pass is changed to an off-screen rendering surface. An off-screen rendering surface behaves like the normal back-buffer that copies the pixel data to the monitor, but an off-screen surface can use another object, e.g., a texture object, to actually store the pixel data. By storing the pixel data to a texture, the first, original rendering pass will put the guessed image in the texture object automatically.

The reference image is also brought in as the second input for the second pass using multitexturing, which allows multiple texture objects to determine the final color of a 3D object. By applying both the reference and guessed images to the same 3D object, the pixel data from both objects are passed as inputs to the pixel shader program. The squaring and summation for each of the color channels of one pixel are programmed into the Direct3D pixel shaders, loading one pixel's color from each texture object and storing the resulting MSE in the pixel's red color channel. The pixel shader engine will automatically execute the pixel shader program for every pixel that is generated in the output of the second rendering pass.

However, the summation inside each pixel shader program can only add the three color channels of that pixel. To perform the summation part of the MSE calculation, we used a rendering feature called alpha blending. Alpha blending performs a weighted summation of the color from a 3D surface visible in one pixel and the 3D surfaces hidden behind the front-most surface to calculate the pixel's final color. This is typically used to create transparent objects by allowing the color of a hidden object to affect the image's final color. To sum the values of every pixel, 3D planes are created, one for every pixel in either of the input images, and placed one behind the other from the perspective of the screen and parallel to each other. Each 3D plane is colored by the result of the pixel shader for one pixel, so that every result is used as the color for a unique 3D plane. Alpha blending will then add the color from each of the 3D planes together and store the result as the color of screen. Since this implementation only requires one value back from the GPU calculation, the screen size is set to one pixel image. When the resulting pixel data is read back to system memory, the red color channel is stored as a 32-bit float and used as the pre-division result of the MSE, which is used to finish the PSNR computation.

C. CUDA Implementation

The CUDA implementation of the MSE calculation is performed in the same way as the shader implementation, but the final summation is implemented using three separate programs.

The first program loads one pixel's data and performs the subtraction and squaring for each of the color channels. The squared distance results are stored in the smaller, but faster shared memory that is accessible to all the threads in a block of threads that are executed together. Since CUDA can execute only up to 512 threads together in one thread block, the results from that thread group are added together and written out as a single value to the graphic card's global memory, which other thread groups can access.

The second program takes the intermediate sum from each thread group, sums those values, and writes that value back to global memory. This summation program executes only if the number of intermediate results is larger than the maximum number of threads per block.

After the intermediate results that can fit within one thread block are computed, the final program adds all the intermediate sums and stores the result in a variable that is read back to system memory. The encoder then finishes the PSNR calculation on the CPU and continues execution.

The reference image is transferred to the graphic memory via CUDA's standard memory copy functions. CUDA has two options to acquire the guessed image from OpenGL. The first method is to copy the image to system memory followed by a copy to CUDA's memory on the graphics card. While detouring the data through system memory is simpler from a coding perspective, it will incur a performance penalty from the slower CPU-GPU link. The second option is to use a pixel buffer object (PBO) to store the image data. PBOs can keep the data on the graphics card's memory and allows CUDA to access the data by mapping the buffer onto its memory space. Both methods are implemented to show the effect of data handling within CUDA's API.

V. Experimental Setup

Since there is no existing simulator for graphics cards or CUDA-capable devices, all testing took place on a Core i7 920 at 2.67 GHz and an nVidia GeForce 9800 GTX+. The rest of the system specs can be found in Table 1. All testing took place in Windows XP using nVidia's 196.21 drivers. All of the programs were compiled using Visual Studio 2005. We used the August 2008 version of the DirectX

Table 1. System Specifications.

L1 Instruction Cache	32 KB per core 4-way associative 64 B lines
L1 Data Cache	32 KB per core 8-way associative 64 B lines
L2 Cache	256 KB per core 8-way associative 64 B lines
L3 Cache	8 MB shared 8-way associative 64 B lines
System Memory	3 GB DDR3 @ 1066 MHz
Chipset	Intel X58

SDK, the OpenGL Utility Toolkit (GLUT) 3.7.6 [23], and the OpenGL Utility Toolkit version 1.5.1 [24] to handle the 3D rendering. CUDA programs are compiled using the CUDA Toolkit version 2.3. Intel's VTune [25] was used to monitor CPU events during execution to show how the processor is utilized during the execution of each program.

The encoder uses a 300-frame clip of a commonly used video in model based video research titled *wow* [19-21]. This lengthened the sampling time sufficiently to reduce variations in the results. After all the testing was completed, there was a maximum of 2% standard deviation on any of the metrics sampled with most variations falling under 1%. As a result, the graphs used show the mean of the samples with no error bars since the error bars would not be visible.

The clip is generated by decoding the (FAPs) from the existing *wow* sequence and saving the frames as bitmap files. The bitmaps, which represent the individual video frames, are then stored in a RAM-disk to remove the overhead of reading files from the hard drive. The size of the rendered image is set to a common video-conferencing resolution of 352x288 (CIF) [26]. The encoder will encode the 300 reference frames using the same model in the image and will exit upon completion of the last frame.

Table 2. Completion time for Encoder (300 frames)

	Total Time (seconds)
D3D-CPU	463.687
D3D-PS	83.562
OGL-CPU	54.312
OGL-CUDA	74.140
OGL-CUDA w/ interop	98.359

VI. Results and Discussion

A. Total Runtime

Table 2 shows the total runtime of each program. The DirectX-based CPU program took 437 seconds to process the 10 seconds of video. When the error calculation is moved to the GPU using the pixel shader, the completion time drops to 83 seconds, which is a 5.25x improvement. However, the OpenGL and CUDA solution that used the OpenGL interoperability (as used in nVidia's CUDA-based OpenGL post-processing example [27]) was slower at 98 seconds, suggesting that it is more efficient to keep the data within the API rather than move it. However, when the image is copied to system memory before moving to CUDA's memory space, a supposedly less efficient method, the total runtime is reduced to 74 seconds. Most surprising is the OpenGL-based program that performed the calculations on the CPU finished in 54 seconds. **Thus, implementations that should be less efficient complete the same computations in less time.**

To find out where the time is being spent, the CPU time from all the CPU cores is broken down by process in Figure 4. While each program was written as a single-threaded application, 3rd party libraries and kernel code could execute in parallel, accounting for the discrepancy in the CPU time

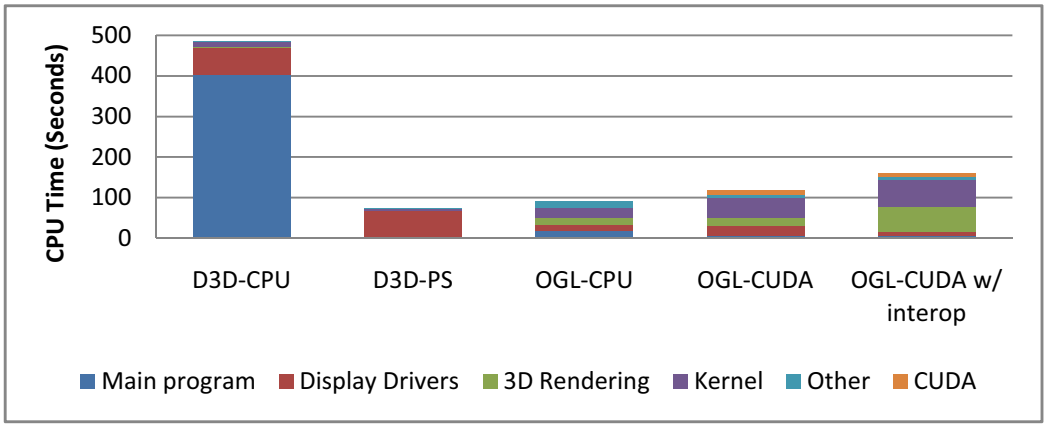


Figure 4. CPU time (based on unhalted cycles)

and total execution time. In the DirectX- and CPU-based code, the majority of the time is spent in the main program. When the error calculation is moved to the pixel shaders, the time spent in the main program drops while the supporting libraries gain a few seconds. On the other hand, the OpenGL programs only see a few seconds spent in the main program. The CUDA-based programs see substantially more time spent in the OpenGL support libraries, CUDA libraries, and the kernel, eclipsing the time reduced from the main program. Based on the increased backend work, CUDA requires a significant amount of time to service its data transfer and computation execution.

Overall, the programs that move the error calculation to the GPU see a drop in the time spent in the main program code, but the required support libraries see additional time to handle the GPU-based computations.

B. Cache Statistics

The cache statistics in Figures 5-7 show some interesting trends with the amount of data handled with each implementation.

The Direct3D pixel shader implementation demonstrates the type of changes that are expected when a data-heavy computation is moved to the GPU. The total number of cache accesses drop drastically, especially in the higher level caches, since the CPU is no longer handling the large image data. There is also a significant decrease in the cache misses per 1000 instructions since the encoder does not have to deal with the image data and its effects on the caches.

However, the OpenGL implementations see either little change or a large increase in the number of cache accesses. While this behavior is expected for the implementation that uses system memory to move the data between OpenGL and CUDA, the CUDA implementation that uses the PBO sees more memory accesses. There is also an increase in misses per 1000 instructions when OpenGL interoperability is used, suggesting that CUDA's OpenGL interoperability adds significant memory overhead to move the data from OpenGL to CUDA itself.

The only execution differences between the OpenGL CPU- and CUDA-based applications was the removal of the CPU-based error calculation and the addition of CUDA-handling code. Since the CPU was no longer handling the error calculation, the CPU should never had to directly

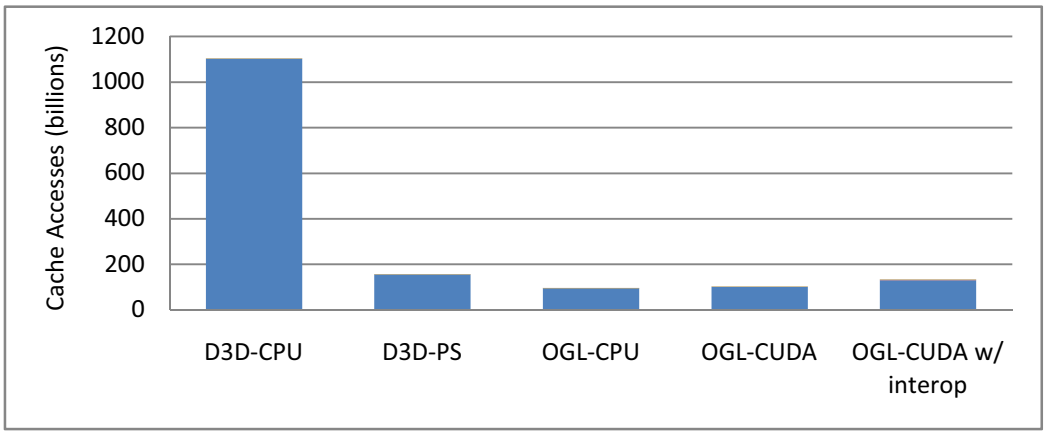


Figure 5. Total data cache accesses by program

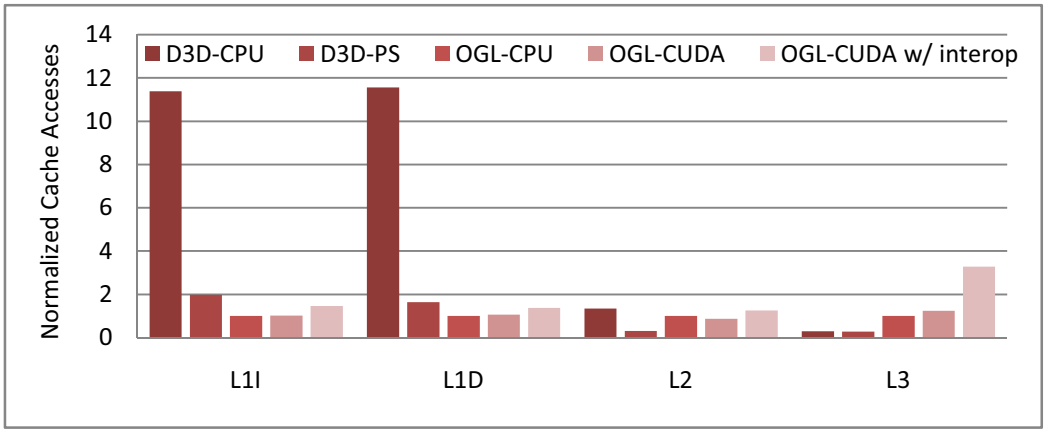


Figure 6. Normalized Cache Accesses (Normalized to OpenGL-CPU)

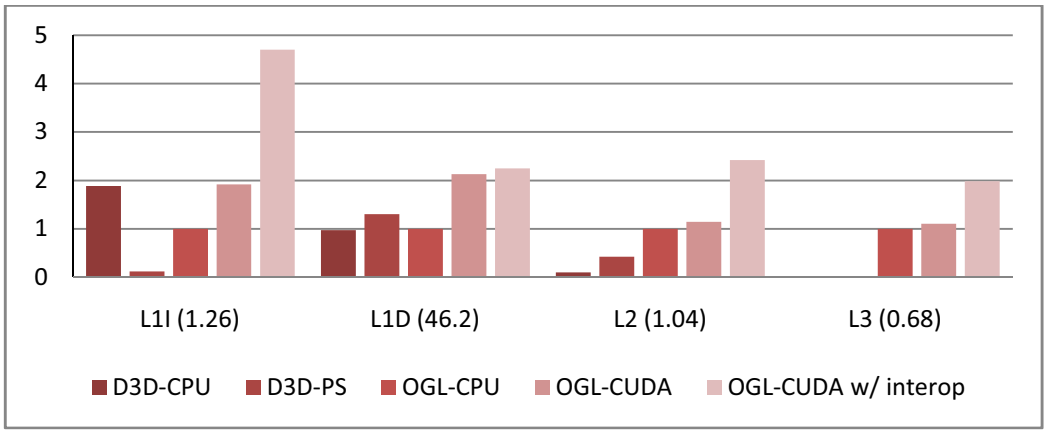


Figure 7. Normalized Misses/1000 Instructions (Normalized to OpenGL-CPU)

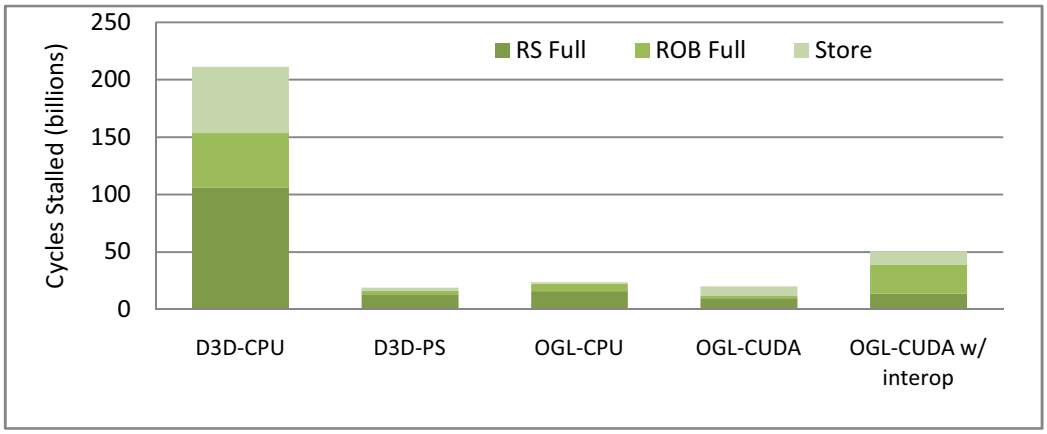


Figure 8. Cycles Stalled by Program

address the pixel data and the cache accesses should have fallen drastically. Instead, cache accesses rose with both CUDA applications. Unless CUDA's libraries are horribly inefficient, the only source of data that CUDA was handling that could have generated as many, if not more accesses, was the data that needed to be copied to CUDA's memory space.

C. Processor Stalls

Finally, an analysis of the processor stalls shown in Figure 8 narrows down the source of CUDA's unexplained behavior.

Between the two Direct3D programs, there is a marked decrease in the two most prevalent stalls, the store buffer stalls and the reservation station stalls. The store unit stall decrease is attributed to the decrease in the amount of data transferred by keeping the hundreds of kilobytes of image data on the graphics card and transferring only the 4-byte result. The reservation station stalls decrease shows the superscalar hardware attempting to parallelize the error calculation due to large number of pixels and the limited number of processing units on the CPU.

Looking to the OpenGL-based programs, the CPU-based implementation sees only a small number of store stalls, despite needing the image data to move to system memory. OpenGL natively handles reading pixel information from a graphics card in the API and transfers the data via direct memory access. Direct3D, on the other hand, requires that data to be copied using the C memcpy function, using the processor to load and store the data. In addition, the Direct3D-CPU version executed over 8 times as many instructions as the OpenGL counterpart, resulting in the dramatic difference in total stalls. The CUDA implementations also have a large number of store unit stalls, suggesting the CUDA-specific CudaMemcpy function also utilized the main processor to move the data to CUDA. This inefficient data-coping method, combined with the cache analysis, suggests that the CUDA libraries are handling the data transfer between the two GPU computation libraries in an inefficient manner.

To verify that the data copy is the utilizing the CPU, the CUDA-based encoders encode the same video at three different input resolutions. The increase in the number of bytes transferred and the number of store units stalls is shown in Figure 9. When the number of store unit stalls is

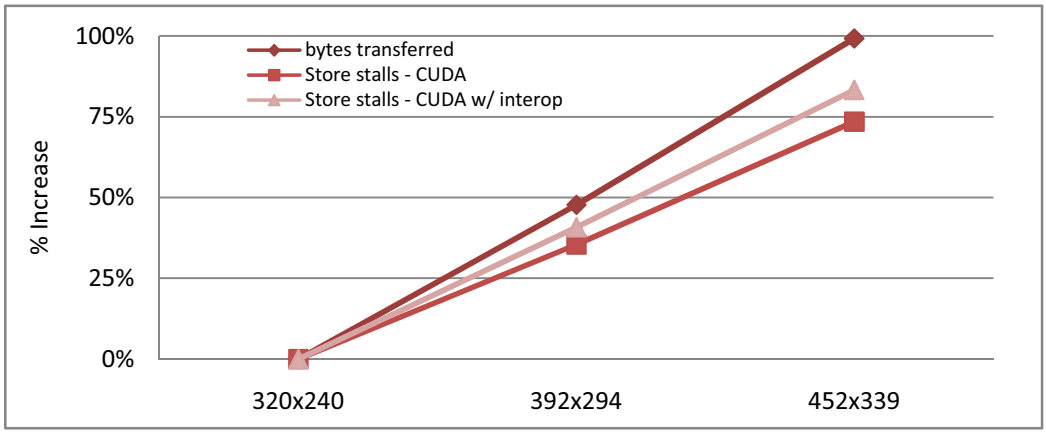


Figure 9. Store Unit Stalls at various resolutions

compared to the amount of data copied, there is a close correlation between the increase of both the input size and the store unit stalls.

In addition, to verify that it is the data copy and not the kernel execution that is generating the store unit stalls, a simple CUDA program was written that executed its kernel 100 times with and without a data copy before the kernel executes. The program that copied the data before the kernel executed saw a 74x increase in the number of store unit stalls. The additional store unit stalls seen in the CUDA program were the result of the image data moving to CUDA controlled memory.

D. Discussion

Based on these findings, it appears that CUDA's libraries must use code running on the main processor to move data between the two graphics card APIs. If there is no option to perform the copy directly on the graphics card and system-level code is required, there appears to be a more efficient method of handling the transfer. The OpenGL implementation that used the CPU to calculate the MSE showed that DMA can handle the data transfer more efficiently because of the shorter total execution time and better resource utilization. Several other researcher assume that CUDA does use DMA [28-29], but do not verify or references sources that state this.

The CUDA libraries handle data copying between system memory and graphics memory using the CPU's load/store unit, as shown by the results in this paper. Requiring the CPU to handle the transfer of several kilobytes of data for every MSE calculation forces the rest of program to wait till the transfer is complete. In addition, this behavior seems to be tied to the presence of the C memcpy function in the code. The Direct3D programs require an explicit memcpy statement to copy data from the graphics card to system memory. In addition, the CUDA APIs have a CUDA-specific memcpy statement that is used to move data to CUDA controlled memory. OpenGL, however, uses DMA to transfer the pixel data to system memory. Since the OpenGL implementation that performed the MSE calculation on the CPU was the fastest and did not use any memcpy functions in the author's code, DMA is a much faster and efficient method for moving data.

In spite of this discovery, we wanted to know if a GPU implementation of the MSE calculation was faster on the GPU than on the main processor. To find out, the two main components of the MSE calculations were timed individually and the entire calculation was timed to show if and where the calculation performed better. Unsurprisingly, the highly parallel subtraction and squaring was 12.9x faster using CUDA, while the highly serial summation was about 5% slower on the GPU. The entire MSE calculation was 70% faster on the GPU than on the main processor, showing that there should have been an improvement if the data-handling problem did not exist.

VII. CONCLUSIONS

While GPGPU computations can provide significant computational speedup over the main system processor, they still suffer from one of the main bottlenecks present in any computer system: *moving data to where it is needed.* Because this is such a well known and debilitating bottleneck, any software solution needs to optimize their data handling to reduce or hide this bottleneck. Here, we performed an analysis of a standard model-based coding system, an application in which data generated on the graphics card using one GPGPU solution is needed as input for another GPGPU API.

Although the initial location and destination reside on the graphics card, nVidia's data-handling within CUDA shows some inefficient trends. Both CUDA implementations showed a net increase in total CPU work and a net increase in the amount of data handled by the CPU. Worse, the integrated OpenGL interoperability provides worse performance and significantly more overhead than if it is not used. If OpenGL and CUDA could exchange data while keeping that data on the graphics card and with minimal

involvement from the CPU, it should not have generated the data observed in this paper.

These problems stem from CUDA's use of the main processor to move data to the graphics card's memory instead of handling the data transfer internally on the graphics card or using the more efficient DMA architecture. Until this is rectified, programs that require large amounts of data to be repeatedly transferred to GPU memory will not find much, if any, improvement in computation time. Programs that exploit the strengths of different types of processors need an efficient method of moving data to justify the use of the specialized processors. Optimizing these critical bottlenecks is required to fully exploit heterogeneous programming, including data sharing between multiple GPGPU APIs.

VIII. Acknowledgements

The authors would like to thank Oklahoma State University for their facilities and OSU's New Product Development Center for the financial support that made this research possible. This work was also supported by the Army Research Office, "Enabling Battlefield Situational Awareness through a Cooperative and Intelligent Video Sensor Network," 56940-CS-DPS. We would also like to acknowledge Eric Larson who wrote the original version of the MBC encoder, on which all of the programs used in this study were based. Finally, we would like to thank Bradley Pesicka for his contributions in C++ and 3D-graphic programming.

IX. References

[1] K. Group. March 5). The Khronos Group: Open Standards, Royalty Free, Dynamic Media Technologies. Available: http://www.khronos.org/

[2] March 29). DirectCompute. Available: http://www.nvidia.com/object/cuda_directcompute.html

[3] NVIDIA. (2009, CUDA Programming Guide 2.3. Available: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[4] Y. Zhiyi, *et al.*, "Parallel Image Processing Based on CUDA," in Computer Science and Software Engineering, 2008 International Conference on, 2008, pp. 198-201.

[5] L. Changxin, *et al.*, "Efficient implementation for MD5-RC4 encryption using GPU with CUDA," in Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on, 2009, pp. 167-170.

[6] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases," presented at the Proceedings of the 2009 IEEE International Symposium on Parallel\&Distributed Processing, 2009.

[7] P. Eisert, *et al.*, "Model-aided coding: a new approach to incorporate facial animation into motion-compensated video coding," Circuits and Systems for Video Technology, IEEE Transactions on, vol. 10, pp. 344-358, 2000.

[8] A. Purde, *et al.*, "Pixel shader based real-time image processing for surface metrology," in Instrumentation and Measurement Technology Conference, 2004. IMTC 04. Proceedings of the 21st IEEE, 2004, pp. 1116-1119 Vol.2.

[9] W. F. Engel, Direct3d Shaderx: Vertex and Pixel Shader Tips and Tricks with Cdrom. Plano, TX: Wordware Publishing Inc., 2002.

[10] W. Engel, ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0. Plano, TX: Wordware Publishing Inc., 2003.

[11] C. Wei-Nien and H. Hsueh-Ming, "H.264/AVC motion estimation implmentation on Compute Unified Device Architecture (CUDA)," in Multimedia and Expo, 2008 IEEE International Conference on, 2008, pp. 697-700.

[12] R. Amorim, *et al.*, "Comparing CUDA and OpenGL implementations for a Jacobi iteration," Universität Graz, Graz, Austria, Technical Report SFB 2008-025, 2008.

[13] K. Kothapalli, *et al.*, "A performance prediction model for the CUDA GPGPU platform," in High Performance Computing (HiPC), 2009 International Conference on, 2009, pp. 463-472.

[14] R. Suda, *et al.*, "Aspects of GPU for general purpose high performance computing," presented at the Proceedings of the 2009 Asia and South Pacific Design Automation Conference, Yokohama, Japan, 2009.

[15] B. R. Coutinho, *et al.*, "Profiling General Purpose GPU Applications," in Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on, 2009, pp. 11-18.

[16] B. Girod, "What's wrong with mean-squared error?," in Digital images and human vision, ed: MIT Press, 1993, pp. 207-220.

[17] T. Wiegand and G. Sullivan, "Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264/ISO/IEC 14\496-10 AVC," vol. JVT-G050, ed, 2003.

[18] A. C. Downton, Speed-up trend analysis for H.261 and model-based image coding algorithms using a parallel-pipeline model vol. 7. Amsterdam, PAYS-BAS: Elsevier, 1995.

[19] F. Lavagetto and R. Pockaj, "An efficient use of MPEG-4 FAP interpolation for facial animation at

70 bits/frame," Circuits and Systems for Video Technology, IEEE Transactions on, vol. 11, pp. 1085-1097, 2001.
[20] I. S. Pandzic, "Facial motion cloning," Graphical Models, vol. 65, pp. 385-404, 2003.
[21] MPEG-4 Facial Animation: The Standard, Implementation and Applications: John Wiley & Sons, Inc., 2003.
[22] M. Doggett, "Programmability Features of Graphics Hardware," 2002.
[23] K. Group. April 14). GLUT - The OpenGL Utility Toolkit. Available: http://www.opengl.org/resources/libraries/glut/
[24] April 14). GLEW: The OpenGL Extension Wrangler Library. Available: http://glew.sourceforge.net/
[25] Intel. March 5). Intel® VTune - Intel® Software Network. Available: http://software.intel.com/en-us/intel-vtune/
[26] I. T. UNION, "H.261 : Video codec for audiovisual services at p x 64 kbit/s," ed, 1994.
[27] NVIDIA. (2010, June 10). NVIDIA CUDA C SDK Code Samples. Available: http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html
[28] J. Breitbart, "Data structure design for GPU based heterogeneous systems," in High Performance Computing & Simulation, 2009. HPCS '09. International Conference on, 2009, pp. 44-51.
[29] A. Kerr, *et al.*, "Modeling GPU-CPU workloads and systems," presented at the Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, Pittsburgh, Pennsylvania, 2010.

Parallelization and Characterization of GARCH Option Pricing on GPUs

Ren-Shuo Liu, Yun-Cheng Tsai, Chia-Lin Yang
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
renshuo@ntu.edu.tw, d98922012@csie.ntu.edu.tw, yangc@csie.ntu.edu.tw

***Abstract*—Option pricing is an important problem in computational finance due to the fast-growing market and increasing complexity of options. For option pricing, a model is required to describe the price process of the underlying asset. The GARCH model is one of the prominent option pricing models since it can model stochastic volatility of the underlying asset. To derive expected profit based on the GARCH model, tree-based simulations are one of the commonly used approaches. Tree-based GARCH option pricing is computing intensive since the tree grows exponentially, and it requires enormous floating point arithmetic operations.**

In this paper, we present the first work on accelerating the tree-based GARCH option pricing on GPUs with CUDA. As the conventional tree data structure is not memory access friendly to GPUs, we propose a new family of tree data structures which position concurrently accessed nodes in contiguous and aligned memory locations. Moreover, to reduce memory bandwidth requirement, we apply fusion optimization, which combines two threads into one to keep data with temporal locality in register files. Our results show 50× speedup compared to a multi-threaded program on a 4-core CPU.

***Index Terms*—GARCH, Mean Tracking, Tree, Option Pricing, GPU, CUDA.**

I. Introduction

Modern 3D graphics processing units (GPUs) have evolved from fixed function pipelines to programmable parallel processors. They do not only demonstrate high rendering throughput for 3D graphics applications. The massive parallelized computing units provided by GPUs also make them the attractive processors to accelerate applications with high data-level parallelism. This is so-called GPGPU [1]. CUDA (Compute Unified Device Architecture) is one of the most successful GPGPU programming techniques [2]. It provides programmers a C language interface to access the GPU hardware and abstracts the underlying GPU architecture. Many application domains, such as electronic design automation (EDA), and medical imaging, have demonstrated great performance potential on GPUs with CUDA implementations [3].

Option pricing in computational finance has also gained a lot of attention for GPGPU computing [3]. Options are a type of contracts written on an underlying asset (e.g., stock). An option holder has right, but not obligation, to buy or sell the asset by a certain date (i.e., maturity date) at a specified price (i.e., strike price). Options give a holder opportunities to profit from the difference between the market price and the strike price. Option pricing is the problem of determining the fair trading price. To determine the fair trading price, we need to predict the expected profit of the option. This requires a stochastic model describing the price process of the underlying asset, and a scheme to derive the expected profit.

Many types of option pricing models and simulation schemes have been developed so far. The pricing models can be classified into constant-volatility and stochastic-volatility models. The former assumes that the volatility (i.e., uncertainty of the asset price) of the underlying asset price remains constant over time. The latter tries to predict the uncertainty in stochastic series for more accurate modeling. Modeling stochastic volatility gives more prediction accuracy, but requires more complicated computation. Most of current studies on GPGPU computing for option pricing focus on constant volatility models. For example, [4] and [5] demonstrate CUDA implementations of the Black-Scholes (BS) model which is a closed-form formula solution to a constant volatility model. [6] and [7] show CUDA implementations of regular binomial lattice and trinomial lattice methods, which are numerical methods applicable to constant volatility models.

In this paper, we focus on option pricing using the GARCH model which can model stochastic volatility of the underlying asset [8]. Tree-based simulations are one of the commonly used approaches to price options [9, 10, 11]. Tree-based methods simulate the evolution of the asset price based on the GARCH model and summarize the profits according to every possible evolution path. Tree-based methods have an advantage of being capable of pricing various types of options. But, tree-based GARCH option pricing is computing intensive since the tree grows exponentially, and it requires enormous floating point arithmetic operations for calculating prices, probabilities, etc.

The tree-based GARCH option pricing exhibits significant data-level parallelism because all tree nodes at the same level are mutually independent. We find that a straightforward CUDA implementation can deliver 5 times speedup over an 8-thread version program on a 4-core CPU, but it causes inefficient memory bandwidth utilization. Traditional tree data structures do not guarantee concurrently accessed data to be located in contiguous and aligned memory locations, which are critical to memory system performance in GPUs. Therefore, we propose a new family of tree data structures that enable efficient memory coalescing. Furthermore, to decrease the memory bandwidth requirement, we use thread fusion to keep data with temporal locality in register files. These optimizations can further boost GPU performance by 10×. Overall, the

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

CUDA implementation of the GARCH option pricing achieves 50× speedup over an 8-thread implementation on a 4-core CPU. Compared to a single-thread CPU implementation, it achieves up to 238× speedup.

The remainder of this paper is organized as follows. Section II describes the architecture of the GPU we used and the tree-based option pricing algorithm. Implementation and optimization details are presented in section III. Experimental setup and performance evaluation results are presented in section IV and V. Section VI discusses related works. Section VII concludes.

II. Background

A. GPU Architecture

In this paper, we target at the NVIDIA's Tesla C1060, which consists of 10 texture processor clusters (TPCs). Each TPC contains 3 streaming multiprocessors (SMs). Each SM contains 8 streaming processors (SPs). Tesla C1060 runs at 1.3 GHz, and delivers 933-GFLOPS peak throughput. In addition to the high computing power, Tesla C1060 has 4 GB global memory, and achieves 102 GB/s of theoretical memory throughput. It contains 16 KB shared memory and 16384 registers per SM. Table I shows more detailed parameters of the C1060 graphic card.

On a GPU, programs are executed in SIMD and fine-grained multi-threading fashion. Threads are statically grouped into warps. For Tesla C1060, a warp is consisted of 32 threads. A warp of threads execute the same instructions, such as an arithmetic operation or a memory access. Furthermore, if one warp is stalled on a long latency instruction, an SM switches context and executes another warp to hide latency.

We use the programming model of NVIDIA's CUDA to develop the tree-based GARCH option pricing. A CUDA program is composed of a host part that executes on a CPU and device parts that execute on GPUs. CUDA hides detail organization of the GPU hardware, such as SM/SP configuration, and treats GPUs as a device that can perform data-parallel algorithms. Moreover, since CUDA uses ANSI-C-like syntax with simple extensions, one can rewrite a C program originally developed on CPUs to CUDA with moderate effort.

TABLE I
NVIDIA Tesla C1060

GPU name	Tesla C1060
# of SM cores	30
Core clock, GHz	1.3
Registers/core	16384
Shared memory/core, KB	16
Memory bus, GHz	0.8
Memory bus, pins	512
Bandwidth, GB/s	102
Memory amount, GB	4
Single Precision, peak Gflop/s	933
Double Precision, peak Gflop/s	78

B. CUDA Optimization Principles

There are three key optimization principles for efficient CUDA implementations. First, in order to fully utilize the massive parallelism provided by GPUs, the target algorithm needs to expose enough data-level parallelism, which often requires restructuring the algorithm. Second, since a group of threads (i.e., a warp in NVIDIA GPU architecture) execute in lockstep in GPUs, reducing divergence among these threads is critical to performance. The last key principle is memory system optimization. Since thousands of threads access the memory subsystem concurrently in GPUs, memory is often the performance bottleneck. Modern GPUs typically support block memory transactions. For example, NVIDIA GPUs have several memory transaction sizes, 32, 64, and 128 bytes. Therefore, if the data accessed concurrently from different threads are located contiguously in memory and aligned to a transaction size, requests can be coalesced into one memory transaction. Therefore, for NVIDIA GPUs, it is critical to keep data accessed in half-warp contiguous and properly aligned [12].

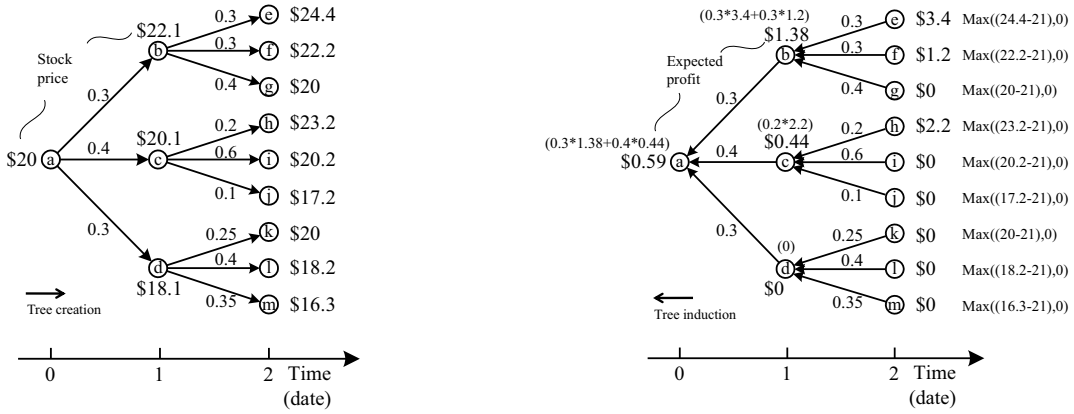
C. Option Pricing

The GARCH option pricing model is a set of stochastic formulas which can predict the movement of the asset price in the future [1]. The MT (Mean Tracking) tree is a numerical method which simulates the GARCH pricing model based on a tree data structure with 3 data fields (price, volatility, and probability) as shown in Figure 1(a) [2]. We use a simple example to show how option pricing works based on the MT-tree algorithm. In Figure 1(a), a stock price is currently \$20 (at date 0), and it is predicted by the GARCH model that at date 1 it will be \$22.1, \$20.1, or \$18.1 with probabilities equal to 0.3, 0.4, and 0.3 respectively. The rest of the tree nodes are constructed in the same way. Given a European call option which allows the purchaser to buy the stock for \$21 at date 2, it will have several possible profit outcomes at date 2. For example, if the stock price turns out to be \$24.4, the profit value of the option will be $\$24.4 - \$21 = \$3.4$ (node ⓔ in Figure 1(a) and 1(b)). If the stock price turns out to be \$20, the value of the option will be \$0, since one will not buy the stock for the contract price (\$21) which is higher than the market price (\$20) (node ⓖ in Figure 1(a) and 1(b)). The option is priced as the probability-weighted sum of all possible profit values. That is, we induct the expected profit at node ⓑ by calculating the probability-weighted sum of profit from node ⓔ, ⓕ, and ⓖ, as Figure 1(b) shows. The rest of the tree nodes are inducted in the same way. Overall, the expected profit value at the root node will be:

$$0.3 \times (0.3 \times \$3.4 + 0.3 \times \$1.2) + 0.4 \times (0.2 \times \$2.2) = \$0.59 \quad (1)$$

Now we summarize the MT-tree implementation in this work. The main data structure of the MT-tree algorithm is a trinomial tree, where each tree node stores three attributes, asset prices, probabilities (how likely the asset price moves to the estimated price), and volatilities (how broadly the estimated price will distribute at the next step). Each attribute is a floating-point number in 4 bytes. The MT-tree algorithm

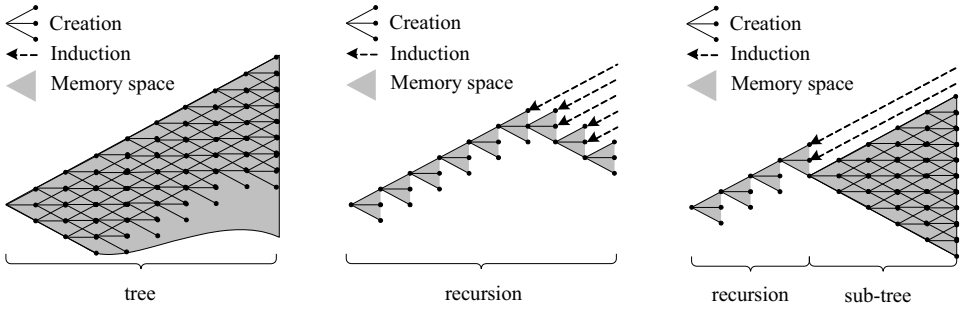
[1] We brief the GARCH model in appendix A.
[2] We brief the MT-tree in appendix B.



(a) Predict the asset price through tree creation

(b) Estimate the value of an European call option (strike price = $21)

Fig. 1. Option pricing using tree-based simulation



(a) Tree (breadth-first) only

(b) Recursion (depth-first) only

(c) Mix recursion and subtree

Fig. 2. Different approaches for tree creation and tree induction

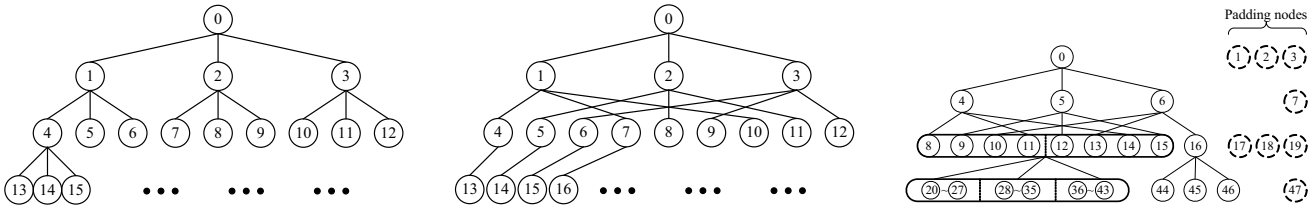
comprises two main stages: **tree creation** and **tree induction**. In the tree creation stage, a tree with predicted asset prices, probabilities, and volatilities in its nodes is created from root node to leaf nodes based on the GARCH model. In the tree induction stage, the expected value of the option is evaluated by evaluating expected profit values from the leaf nodes and working backward to the root.

III. Parallelization Methodology

In the MT-tree algorithm, tree nodes at the same level are independent. Furthermore, the number of tree nodes grows exponentially with the number of maturity days. Therefore, we can exploit the massive parallelism provided by GPUs by processing tree nodes at the same level concurrently. In this section, we first go over the tree traversal method. We then introduce the optimization techniques that we adopt to improve memory system performance, including new tree data structure and thread fusion. Finally, we describe our CUDA implementation details.

A. MT-Tree Traversal Method

There are two ways to traverse the MT-tree, breadth-first and depth-first approaches. As Figure2(a) shows, the breadth-first approach constructs the tree nodes level by level till the leaf level is reached. Then it performs backward induction level by level in reverse to calculate the option value at the root. It exposes the most amount of data-level parallelism since all nodes at the same level are independent. The drawback of breadth-first simulation is that it requires large memory space to store the whole tree before tree induction. On the other hand, as shown in Figure2(b), the depth-first approach constructs the tree in one branch, and then performs tree induction as soon as possible. Compared to the bread-first approach, it requires smaller memory space but it exposes less parallelism. Therefore, to expose enough parallelism with reasonable memory space, as shown in Figure 2(c), we construct the tree in the depth-first manner to a certain level and then adopt the breadth-first approach to grow the whole subtree.



(a) Traditional array implementation of trees

(b) Consecutive array implementation of trees (C-Tree)

(c) Coalesced and aligned array implementation of trees (CA#-Tree)

Fig. 3. Tree Data Structures

B. MT-tree Data Structure and Optimization

There are two typical tree data structures, array and link representations [13]. Since the MT-tree algorithm does not need to insert or delete tree nodes, we use arrays to store the subtree in Figure 2(c). The conventional array representation of a tree is shown in Figure 3(a), where each tree node is denoted with its index in a one-dimensional array. This tree data structure works well on CPUs but causes dramatic performance losses on GPUs. The main issue is that the memory accesses from concurrent threads (i.e., a warp) are not contiguous and properly aligned. For example, as mentioned earlier, a MT-tree node is a record with 3 float fields, price (y), volatility (h), and probability (p). Since a warp operates on the same field of multiple nodes together, these accesses are not contiguous as shown in Figure 4. Another example is when the threads in a warp operate on node 1, 2, and 3, their outputs are placed at node 4, 7, and 10, which again are not contiguous memory accesses.

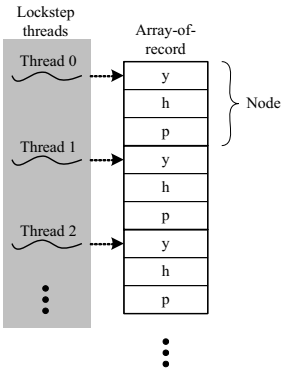


Fig. 4. Memory access inefficiency due to "array of record"

As discussed in Section II-B, the key principle to optimize GPU memory system performance is to keep memory accesses from the same half-warp contiguous and properly aligned. To solve the first issue which is caused by "array of records", [14] suggests to store different data fields into different arrays. To tackle the second problem, it is essential to redesign the tree data structure. In this paper, we propose a new family of tree data structures, C-Tree and CA#-Tree.

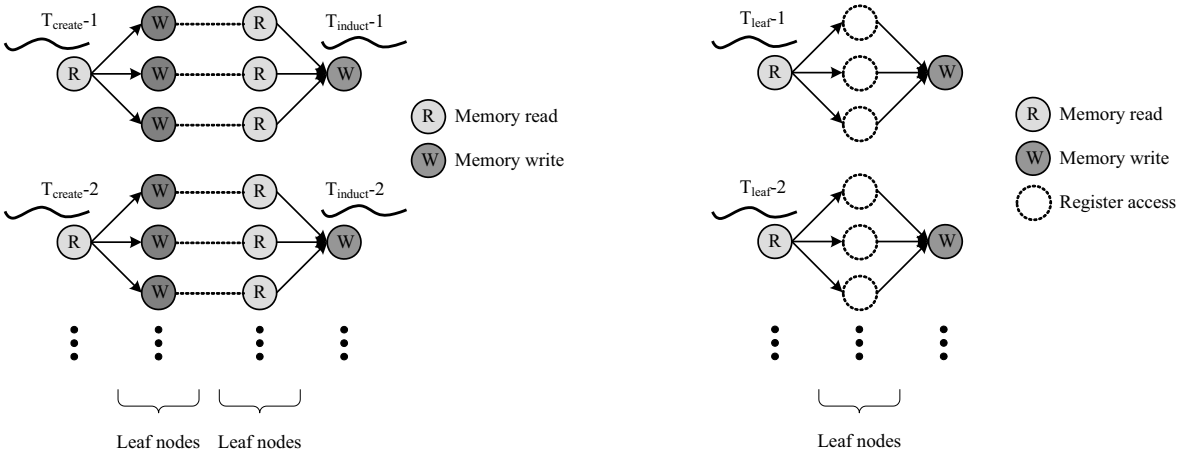
C-Tree (Consecutive Tree) is shown in Figure 3(b). In contrast to the traditional tree representation, in C-Tree, the leftmost (middle/rightmost) child nodes of tree nodes from the same level are placed contiguously in memory. For example, the left children of nodes 1, 2, and 3 are node 4, 5, and 6 instead of node 4, 7, and 10 in the traditional tree. In this way, all concurrent threads read and write *consecutive* memory addresses.

CA#-Tree (Consecutive and Aligned to #byte Tree) improves C-Tree by further considering the alignment issue. Let's take CA16-Tree which aligns memory accesses to 16 bytes as an example. First, to ensure that each tree level starts with aligned addresses, the number of tree nodes at each tree level needs to be multiple of 4 since each tree node contains 4 bytes $(4 \times 4 = 16)$. Therefore, in each level, we insert dummy nodes to satisfy this requirement as shown in Figure 3(c). Moreover, since tree creation and induction access parent nodes and child nodes at the same time, the parent nodes and the child nodes accessed by a half-warp should also form aligned transactions. To achieve this, all nodes at the same level are viewed as 2 groups, a coalesced group and a remainder group. The coalesced group is those nodes which form multiple complete and aligned transactions. For example, at level 2, node 8 to 15 form the coalesced group, and node 16 is in the remainder group. To construct the next level, instead of creating all leftmost (middle/rightmost) children together as in C-Tree, CA16-Tree creates the children of a coalesced group first. Tree nodes in a remainder group are then handled separately. In this way, we can guarantee that all parent and child node accesses in a coalesced group are all properly aligned.

In CA#-Tree structures, most tree nodes are in coalesced groups. Let S be the transaction size, N be the total number of tree nodes. Since there are at most $S-1$ nodes which are not in the coalesced group at each level. Therefore, the remainder group presents the following percentage of the whole tree:

$$\frac{\log_3(N) \times (S-1)}{N} \tag{2}$$

This is an exponentially small fraction given the constant S. Therefore, the CA#-Tree is almost perfect-coalesced.



(a) Before Fusion Optimization

(b) After Fusion Optimization

Fig. 5. Fusion Optimization

C. Thread Fusion Optimization

The goal of thread fusion is to keep data with temporal locality in register files to reduce memory accesses. We apply fusion optimization to the threads operating on the leaf nodes. Without thread fusion, as shown in Figure 5(a), there are separate threads handling leaf node creation and induction. In this example, T_{create} and T_{induct} read/write 16 nodes from global memory. The proposed fusion optimization is shown in Figure 5(b). Leaf node creation and induction are performed in the same threads, T_{leaf}. In this way, all the leaf nodes can be kept in register files in contrast to Figure 5(a) where leaf nodes need to be written to global memory first, and then read back again by induction threads. Thread fusion can effectively reduce memory bandwidth requirement of the MT-tree algorithm.

D. Implementation Details

The resulting thread programs are outlined in Algorithm. 1, 2, 3, and 4.

Algorithm. 1 is the host part of the CUDA program to handle a subtree. Let seg be the transaction size in unit of nodes, $branch$ be the number of branches of the tree. The program first creates the subtree level by level. In the program, $readBase$ keeps pointing to the left-most node of the current tree level. $writeBase$ keeps pointing to the left-most node of the level which depends on the on-going computations. a keeps the number of nodes in the coalesced group at current level. b keeps the number of nodes in the remainder group at current level. At the leaf level of the tree, the host invokes another GPU kernel, treeLeaf(), which adopts the fusion optimization. Then, the program inducts the value of option level by level from leaf to root.

Algorithm 2 shows the GPU kernel which can parallelly perform tree creating tasks. In the program, each thread first obtains a thread index, then loads source data from global memory to its registers. Every thread invokes GarchCreateStep(), which is corresponding to the GARCH modeling equations, to compute prices, volatilities, and probabilities. Lastly, the threads store the computation results back to global memory. We present the array indexing formulas for CA#-Tree in the code.

Algorithm 3 is the GPU kernel responsible for processing the leaf nodes of the tree. In this kernel, threads invoke treeInduct() right after treeCreate() to improve the data locality in register files. The treeInduct() function is related to the type of option to be valued. Take European option for example, the treeInduct() function computes probability-weighted sum.

Algorithm 4 is the GPU kernel which can parallelly perform tree induction jobs. Its procedure is similar to the one in Algorithm 2 but in reverse.

The number of threads per block for CUDA kernels is set to 128. This value is selected through experiments. The block number is determined at runtime using the equation: $Min(ceiling((a+b)/128), 512)$.

IV. Experimental Setup

We perform GPU experiments on NVIDIA Tesla C1060 GPU as described in Section II-A. We implement 7 versions of CUDA programs to understand the effects of different optimization techniques described in Section III. GPU-Base is a basic SIMD version which uses the traditional tree data structure. GPU-Split separates array-of-records into 3 arrays which store different data fields separately. In addition to separating array-of-records into 3 arrays, GPU-C-Tree adopts the C-Tree data structure (Figure 3(b)), and GPU-CA32-Tree/GPU-CA64-Tree/GPU-CA128-Tree align each level of tree to 32, 64 and 128 bytes boundary respectively (Figure 3(c)). Lastly, GPU-Fusion is based on GPU-CA128-Tree and further adopts thread fusion technique to reduce memory access amount.

Algorithm 1 Host code for tree-based GARCH option pricing

$a = 0$
$b = 1$
$writeBase = 0$
for $i = 1$ **to** $treeHeight - 1$ **do**
  $readBase = writeBase$
  $writeBase = writeBase + ceiling((a+b)/seg) * seg$
  treeCreate$(a, b, readBase, writeBase)$
  $a = floor((a + b) * branch/seg) * seg$
  $b = b * branch\%seg$
  threadSync()
end for
$readBase = writeBase$
treeLeaf$(a, b, readBase, writeBass)$
$writeBase = writeBase + ceiling((a + b)/seg) * seg$
$swap(readBase, writeBase)$
$a = floor(power(branch, treeHight - 1)/seg) * seg$
$b = power(branch, treeHight - 1)\%seg$
threadSync()
for $i = treeHeight - 1$ **to** 1 **do**
  treeInduct$(a, b, readBase, writeBase)$
  $b = ((a + b)/branch)\%seg$
  $a = a/(branch * seg) * seg$
  $readBase = writeBase$
  $writeBase = writeBase - ceilling((a+b)/seg) * seg$
  threadSync()
end for

Algorithm 2 GPU kernel: treeCreate()

$tid = blockIdx.x * blockDim.x + threadIdx.x$
while $tid < a + b$ **do**
  read node$[readBase + tid]$
  GarchCreateStep()
  if $tid < a$ **then**
    for $k = 0$ **to** $branch - 1$ **do**
      write node$[writeBase + a * k + tid]$
    end for
  else
    for $k = 0$ **to** $branch - 1$ **do**
      write node$[writeBase + a * (branch - 1) + b * k + tid]$
    end for
  end if
  $tid = tid + gridSize$
end while

Algorithm 3 GPU kernel: treeLeaf()

$tid = blockIdx.x * blockDim.x + threadIdx.x$
while $tid < a + b$ **do**
  read $node[readBase + tid]$
  GarchCreationStep()
  GarchInductionStep()
  write $node[writeBase + tid]$
  $tid = tid + gridSize$
end while

Algorithm 4 GPU kernel: treeInduct()

$tid = blockIdx.x * blockDim.x + threadIdx.x$
while $tid < a + b$ **do**
  if $tid < a$ **then**
    for $k = 0$ **to** $branch - 1$ **do**
      read node$[readBase + a * k + tid]$
    end for
  else
    for $k = 0$ **to** $branch - 1$ **do**
      read node$[readBase + a * (branch - 1) + b * k + tid]$
    end for
  end if
  GarchInductionStep()
  write node$[writeBase + tid]$
  $tid = tid + gridSize$
end while

We implement both sequential and parallelized C version programs for comparison between CPUs and GPUs. The arithmetic kernels (i.e., the C code which performs single step of tree-creation or tree-reduction) are the same for both CPU and GPU programs. The C-version ones use traditional array representation of trees as shown in Figure 3(a), and each node in the tree is a record with 3 float data fields. The parallelized version utilizes OpenMP, a widely-used parallel programming API. The OpenMP scheduling policy is static scheduling with the chunk size equal to 10000. This setting is decided through experiments. The compiler we use is GCC 4.3.2 with the highest compiling optimization level (-O3) and targets x86_64 architecture. We perform experiments on a state-of-the-art multi-core system consisting an Intel Core i7 920 (2.66GHz, 4-core, 8-thread SMT) and 6GB (3 channels×2GB) DDR3 1600 main memory.

We perform a 20-day tree simulation (5-day recursive and 15-day subtree) for performance benchmarking. The parameters setup of the option pricing problem is shown in Table II, which is obtained from [9].

V. Evaluation

In this section, we first examine the execution time breakdown of the GARCH simulation on a CPU for understanding its computation complexity. We then analyze its performance on GPUs considering different optimizations described in Section III, including performance speedup, memory bandwidth, and memory transaction profiles. Finally, we conduct a numerical evaluation to demonstrate the soundness of our parallelized algorithms.

A. Execution Time Break Down Analysis

Figure 6 shows the execution time of tree creation and tree induction on CPUs for both single-thread and 8-thread versions. The execution time is broken down into memory accesses and computation. To roughly estimate the memory access time, we replace tree creation and induction step with dummy arithmetic routines to measure the execution time. We

can first observe that for the single-thread version, both tree creation and induction processes are computation bound. Only 4.5% of execution time is due to memory accesses. The 8-thread version on 4-core CPU achieves 5× speedup. We can see that the GARCH option pricing indeed exhibits massive parallelism and is suitable for GPU acceleration.

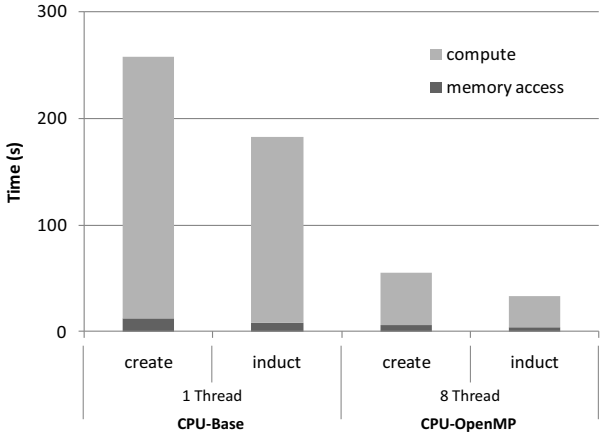


Fig. 6. Breakdown of execution time on CPU

B. GPU Performance

Figure 7 analyzes the GPU performance in terms of normalized performance and effective memory bandwidth. We also show CPU performance for comparison. CPU-Base and CPU-OpenMP correspond to single-thread and 8-thread CPU implementations. We can see that the straightforward GPU implementation, GPU-Base, is 5 times faster than CPU-OpenMP. We can also observe that the delivered memory throughput of GPU-Base is 9.5 GB/s which is far from the peak bandwidth (76 GB/s) the C1060 can typically offer [14]. GPU-Split which stores 3 data fields of a record in 3 trees can further boost GPU performance by 2.2 times. The performance advantage comes from more efficient bandwidth utilization. We can observe that the effective bandwidth of GPU-Split is higher than GPU-Base, 21.5 GB/s vs. 9.5 GB/s. Note that although the MT-tree algorithm appears to be computation-bound on CPUs, it becomes memory-bound on GPUs since thousands of threads access the memory subsystem simultaneously.

Next, we examine the effects of the new tree data structures proposed in this paper, C-Tree and CA#-Tree. We can see that C-Tree, which ensures the data accessed by concurrent threads to be contiguous, gains additional 2× speedup over GPU-Split. It doubles the effective memory bandwidth (40.7 GB/s. vs. 21.5 GB/s). For CA#-Tree, since NVIDIA C1060 coalesces contiguous global memory accesses into 32-byte, 64-byte, or 128-byte transactions. We examine three alignment sizes, 32-byte, 64-byte, or 128-byte (GPU-CA32-Tree, GPU-CA64-Tree and GPU-CA128-Tree). The experimental results show that they make an additional 1.5× speedup over GPU-C-Tree. GPU-CA128-Tree is slight faster than GPU-CA64-Tree. It achieves 64 GB/s of bandwidth which is very close to the peak bandwidth typically delivered by C1060 (76 GB/s)[14].

The last optimization we examine is thread fusion. The experimental results show that thread fusion (i.e., GPU-Fusion in Figure 7) achieves additional 2× speedup over GPU-CA128-Tree. The performance advantage comes from reducing memory accesses by keeping data with temporal locality in register files. We can see that the effective bandwidth becomes 43.7 GB/s compared to 64GB/s of GPU-CA128-Tree. This is an indication that with the thread fusion optimization, GPU performance is no longer bounded by memory performance. Overall, the CUDA implementation of the MT-tree option pricing algorithm is 50 times faster than CPU-OpenMP, and 238 times faster than CPU-Base.

C. Memory Transaction Profile

To further analyze the memory system behavior, Figure 8 shows the number of memory transactions issued by one TPC of the GPU for different CUDA implementations. These data are gathered using NVIDIA's CUDA Visual Profiler utility. The transactions are classified into 6 categories by their types (i.e., load or store) and sizes (i.e., 32 byte, 64 byte, or 128 byte). We can see that GPU-Base presents the most transaction counts, and also has the most 128-byte transactions. Since for Tesla C1060, there are 16 threads in a half-warp, and our program accesses 4-byte data per thread. Those 128-byte transactions indicate a lot of non-coalesced memory accesses and waste of memory bandwidth. The experimental results show that with the optimization techniques proposed in this paper, memory transaction counts are significantly reduced. We can see that most of memory transactions of GPU-CA64-Tree are 64 bytes. This is the most compact transaction type for our program. Moreover, the GPU-CA128-Tree presents identical transaction profile to GPU-CA64-Tree because a 128-byte-aligned tree is 64-byte-aligned as well. GPU-Fusion further reduces the number of 64-byte memory transaction since it can keep data with temporal locality in register files. Compared to GPU-CA128-Tree, GPU-Fusion reduces total memory transaction both in counts and traffic size by 2.2 times.

D. Numerical Evaluation Call Option Prices

To show that our parallelized GARCH option pricing implementation does not incur errors, Table III evaluates the pricing results of European call options for different strikes and maturities using the CPU-Base program, the GPU-Fusion program, and Monte-Carlo simulation. The 30-day results for CPU-Base are not shown because their execution time are too long. The results show that the optimization methods and the CUDA implementations in Section III are correct, since the pricing results from CPU-Base and GPU-Base are equal. Moreover, it is a typical verification method to compare the pricing results from tree simulation to the ones form Monte-Carlo simulation. The verification results also show that the pricing results from CPU-Base and GPU-Fusion are reasonable since they both within or closed to the 95% confident intervals from Monte-Carlo based simulation.

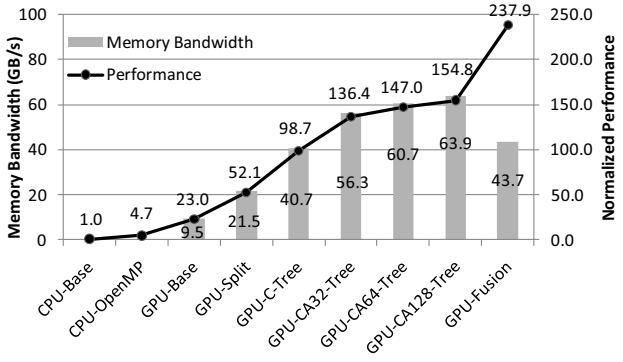


Fig. 7. Performance of computation on GPUs

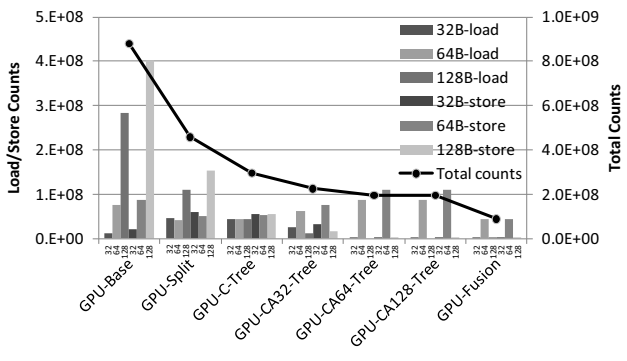


Fig. 8. Memory access profile on GPUs

VI. Related Work

Cluster computers have been used to accelerate option pricing for a long time. Freedman et al. [15], Chen et al. [16], and Kim et al. [17] all propose to use cluster computers to perform Monte-Carlo option pricing. Recently, instead of cluster computers, many works have used GPUs to accelerate option pricing. Podlozhnyuk [4] proposes to use GPUs to price European options in the BS model. In addition to the BS model, Podlozhnyuk [7], Jauvion et al. [6], and Pharr et al. [5] provide several GPU implementations of binomial or trinomial lattice. Furthermore, Pharr et al. [5], Lee et al. [18] and Abbas-Turki et al. [19] present GPU-based Monte-Carlo option pricing. These option pricing methods are different from the tree-based method we focus in this paper. The BS model is a closed-form formula. In contrast to the lattice-based computation which can store data in a linear array for GPU computation, the tree data-structure presents the non-coalesced memory access issue on GPUs before optimization. Lastly, the Monte-Carlo-based option pricing is based on random sampling. Most of the Monte-Carlo-based works focus on designing an efficient random number generator. They are different from the tree-based simulation. The Cell processor is another interesting multi-core architecture [20]. Agarwal et al. [21] demonstrate using the Cell processor to parallelize Monte-Carlo-based option pricing. The results show a speedup of 1.51 over an NVIDIA GeForce 8800 graphics card.

Accelerating option pricing through dedicated hardware has also been widely studied. Zhang et al. [22], Morris et al. [23], Thomas et al. [24], Tian et al. [25], and Castillo et al. [26] implement the Monte-Carlo method using FPGAs. Jin et al. [27] develop FPGAs to perform option pricing using binomial lattice and trinomial lattice.

Tree approaches have been used to parallelize computations on GPUs, such as parallel reduction [28] and parallel prefix sum [29]. The former computes the sum of large arrays of values, and the latter computes the accumulation up to every index of large arrays of values. They adopt trees and optimize memory access efficiency on GPUs as well. But the trees are the schedules of operations instead of data structures. Therefore, they are different from the C-Tree and CA#-Tree proposed in this work.

Data transfer sizes have been considered in many tree data structures design. For example, B-tree [30] and B+ tree [31] increase the branching factor of the trees so that the size of each node corresponds to the data transfer size of the underlying storage device (i.e., disks). This simplifies and optimizes disk accesses. B-tree and and B+ tree target at applications which require efficient searching, insertion, and deletion. But they are not suitable for the computation flow of tree-based GARCH option pricing studied in this work.

VII. Conclusion

In this paper, we present the first characterization and parallelization work of GARCH option pricing using tree-based simulations on GPUs. We find that the conventional tree data structure is not memory access friendly to GPUs. Therefore, we propose the C-Tree and CA#-Tree data structures which provide efficient memory accesses on GPUs. Moreover, we use thread fusion to keep data with temporal locality in register files. This can further reduce memory accesses. Our results show that, after optimizations, the CUDA implementation on Tesla C1060 graphics card achieves 50× speedup compared to a multi-threading quad-core CPU. The numerical evaluation also shows the accuracy of the results from GPUs.

Acknowledgments

The authors would like to thank the insightful comments and constructive suggestions from anonymous reviewers. This research is supported in part by research grants from ROC National Science Council NSC 99-2220-E-002-027-, NSC99-2000-E-002-021, Excellent Research Projects of National Taiwan University 99R80304, Etron Technology Inc., 99R70500, and MacronixInternational Co., Ltd. 99-S-C25.

References

[1] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn, "Gpgpu: general purpose computation on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, New York, NY, USA, 2004, p. 33, ACM.

TABLE II
PARAMETERS OF THE OPTION PRICING PROBLEM

Parameter	Value
S_0	100
r	0
h_0^2	0.0001096
β_0	0.000006575
β_1	0.9
β_2	0.04
c	0

TABLE III
CALL OPTION PRICES FOR DIFFERENT STRIKES AND MATURITY

Strike Price k	Maturity (days)								
	10			20			30		
	GPU	CPU	Monte Carlo	GPU	CPU	Monte Carlo	GPU	CPU	Monte Carlo
95.0	5.081	5.081	(5.080, 5.094)	5.315	5.315	(5.310, 5.326)	5.567	--	(5.563, 5.582)
97.5	2.913	2.913	(2.909, 2.920)	3.354	3.354	(3.342, 3.356)	3.722	--	(3.704, 3.721)
100.0	1.313	1.313	(1.305, 1.313)	1.861	1.861	(1.848, 1.859)	2.281	--	(2.273, 2.287)
102.5	0.436	0.436	(0.436, 0.441)	0.893	0.893	(0.886, 0.894)	1.273	--	(1.267, 1.277)
105.0	0.098	0.098	(0.109, 0.111)	0.363	0.363	(0.368, 0.373)	0.640	--	(0.642, 0.649)

[2] David Kirk, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM '07: Proceedings of the 6th international symposium on Memory management*, New York, NY, USA, 2007, pp. 103–104, ACM.

[3] "Cuda zone," http://www.nvidia.com/object/cuda_home_new.html.

[4] Victor Podlozhnyuk, "Black-scholes option pricing," .

[5] Matt Pharr and Randima Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, Addison-Wesley Professional, 2005.

[6] Gregoire Jauvion and Ecole Centrale Paris, "Parallelized trinomial option pricing model on gpu with cuda," http://www.arbitragis-research.com/cuda-in-computational-finance/.

[7] Victor Podlozhnyuk, "Binomial option pricing model," .

[8] Jin-Chuan Duan, "The garch option pricing model," *Mathematical Finance*, vol. 5, no. 1, pp. 13–32, 1995.

[9] Peter Ritchken and Rob Trevor, "Pricing options under generalized garch and stochastic volatility processes," *Journal of Finance*, vol. 54, no. 1, pp. 377–402, 02 1999.

[10] N. Cakici and K. Topyan, "The garch option pricing model: a lattice approach," *Journal of Computational Finance*, vol. 3, no. 4, pp. 71–85, 2000.

[11] Yuh-Dauh Lyuu and Chi-Ning Wu, "On accurate and provably efficient garch option pricing algorithms," *Quantitative Finance*, vol. 5, no. 2, pp. 181–198, 2005.

[12] NVIDIA, "Nvidia cuda programming guide, v2.3," 2009.

[13] Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed, *Fundamentals of Data Structures in C*, Silicon Press, Summit, NJ, USA, 2007.

[14] Paulius Micikevicius, "Cuda optimization," .

[15] R.S. Freedman and R. Di Giorgio, "New computational architectures for pricing derivatives," in *Computational Intelligence for Financial Engineering, 1996., Proceedings of the IEEE/IAFE 1996 Conference on*, 24-26 1996, pp. 14 –19.

[16] Gong Chen, P. Thulasiraman, and R.K. Thulasiram, "Distributed quasi-monte carlo algorithm for option pricing on hnows using mpc," in *Simulation Symposium, 2006. 39th Annual*, 2-6 2006, p. 8 pp.

[17] Jin Suk Kim and Suk Joon Byun, "A parallel monte carlo simulation on cluster systems for financial derivatives pricing," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, 2-5 2005, vol. 2, pp. 1040 – 1044 Vol. 2.

[18] Myungho Lee, Chin Hong Chun, and Sugwon Hong, "Financial derivatives modeling using gpu's," *Scalable Computing and Communications; International Conference on Embedded Computing, International Conference on*, vol. 0, pp. 440–445, 2009.

[19] Lokman A. Abbas-Turki, Stephane Vialle, Bernard Lapeyre, and Patrick Mercier, "High dimensional pricing of exotic european contracts on a gpu cluster, and comparison to a cpu cluster," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2009, pp. 1–8, IEEE Computer Society.

[20] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[21] V. Agarwal, Lurng-Kuo Liu, and D.A. Bader, "Financial modeling on the cell broadband engine," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 14-18 2008, pp. 1 –12.

[22] G.L. Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi, C.C.C. Cheung, D.-U. Lee, R.C.C. Cheung, and W. Luk, "Reconfigurable acceleration for monte carlo based financial simulation," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, 11-14 2005, pp. 215 –222.

[23] G.W. Morris and M. Aubury, "Design space exploration of the european option benchmark using hyperstreams," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 27-29 2007, pp. 5 –10.

[24] D.B. Thomas, J.A. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial monte-carlo simulations," in *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, 9-11 2007, pp. 168 –173.

[25] Xiang Tian and K. Benkrid, "Design and implementation of a high performance financial monte-carlo simulation engine on an fpga supercomputer," in *ICECE Technology, 2008. FPT 2008. International Conference on*, 8-10 2008, pp. 81 –88.

[26] J. Castillo, Jose L. Bosque, E. Castillo, P. Huerta, and J. I. Martinez, "Hardware accelerated montecarlo financial simulation over low cost fpga cluster," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2009, pp. 1–8, IEEE Computer Society.

[27] Qiwei Jin, David B. Thomas, Wayne Luk, and Benjamin Cope, "Exploring reconfigurable architectures for tree-based option pricing models," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 4, pp. 1–17, 2009.

[28] Mark Harris, "Optimizing parallel reduction in cuda," .

[29] Hubert Nguyen, *Gpu gems 3*, Addison-Wesley Professional, 2007.

[30] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *SIGFIDET '70: Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, New York, NY, USA, 1970, pp. 107–141, ACM.

[31] Douglas Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.

[32] John C. Hull, *Options, Futures, and Other Derivatives*, Financial Times, 4th edition, 1999.

APPENDIX

A. The GARCH Model

Here we briefly describe the GARCH model (Generalized AutoRegressive Conditional Heteroskedasticity). GARCH is one of the financial models which can help estimating the movement of asset prices. It is emerging due to its ability to capture the serial-correlating nature of volatilities (variance of asset price movement). At date t, let S_t be the asset price, $y_t = lnS_t$ be the logarithmic price, and h_t be the volatility over the date [t, t+1] period. Given information at date t, y_{t+1} and h_{t+1} will follow the following process [8]:

$$\epsilon_{t+1} \sim N(0,1), \tag{3}$$

$$h_{t+1}^2 = \beta_0 + \beta_1 h_t^2 + \beta_2 h_t^2 (\epsilon_{t+1} - c)^2, \tag{4}$$

$$y_{t+1} = y_t + r - \frac{h_t^2}{2} + h_t \epsilon_{t+1}. \tag{5}$$

Where ϵ_{t+1} is a standard normal random variable. r is the daily riskless return (e.g., interest rate). c represents a negative correlation between the shock for the asset return and its volatility. β_0, β_1 and β_2 are regression coefficients from history.

B. The MT-Tree Algorithm

The GARCH model is stochastic and continuous. Therefore, it does not describe a tree itself. The MT-tree algorithm can create a tree as the one in Figure 1(a) in the GARCH model.

MT-tree [11] is an improved trinomial tree over RT tree [9] and CT tree [10] in the GARCH model. In MT-tree algorithm, every tree node contains 3 attributes, asset price (denoted as y), volatility (denoted as h), and probability (denoted as p). Furthermore, every node has 3 successor nodes which stand for 3 possible events that the asset price moves upward, middle, or downward. MT-tree lets the middle branch track the mean price of the underlying asset to improve the accuracy.

According to MT-tree, given the node at date t, the 3 successor nodes can be created using the following formulas:

$$l = 0, \pm 1, \tag{6}$$

$$\gamma_n = \frac{\sqrt{min(h_0^2, \frac{\beta_0}{1-\beta_1})}}{2\sqrt{n}}, \tag{7}$$

$$\mu = r - \frac{h_t^2}{2} \tag{8}$$

$$\eta = \left\lceil \sqrt{\frac{n h_t^2 + (a\gamma_n - \mu)^2}{n\gamma_n}} \right\rceil, \tag{9}$$

$$a \in Z \quad , \quad \frac{\gamma_n}{2} \geq |a\gamma_n - \mu| \tag{10}$$

$$\epsilon_{t+1} = \frac{l\eta\gamma_n + a\gamma_n - \mu}{h_t}, \tag{11}$$

$$y_{t+1} = y_t + l\eta\gamma_n + a\gamma_n, \tag{12}$$

$$h_{t+1} = \sqrt{\beta_0 + \beta_1 h_t^2 + \beta_2 h_t^2 (\epsilon_{t+1} - c)^2}, \tag{13}$$

$$p_{t+1} = \begin{cases} 1 - \frac{n h_t^2 + (a\gamma_n - \mu)^2}{n^2\eta^2\gamma_n^2}, & \text{if } l = 0 \\ \frac{n h_t^2 + (a\gamma_n - \mu)^2}{2n^2\eta^2\gamma_n^2} - l\frac{a\gamma_n - \mu}{2n\eta\gamma_n}, & \text{otherwise} \end{cases} \tag{14}$$

In equation (6), l takes values of -1, 0, and 1 for the 3 successor nodes (upward, middle, and downward nodes respectively). Furthermore, equations (7) to (11) compute intermediate variables using the parameters of the GARCH model, such as β_0, β_1, β_2, r, and c. Given current state y_t and h_t, equations (12) to (14) calculate the tree attributes y_{t+1}, h_{t+1}, and p_{t+1} for the 3 successor states. The full MT-tree grows exponentially since the number of possible volatilities h_t at a node is exponentially many. The approximated MT-tree grows quadratically under certain threshold condition[11].

Once the tree is created, we can price options by calculating the expected value. For European call option, we can just compute the probability-weighted sum as described in Section II-C. For other option types, such as American options which allow the purchaser to early excise the option contract, the computation will be a little different. For more details, please refer to [32].

Characterization of Workload and Resource Consumption for an Online Travel and Booking Site

Nicolas Poggi*‡, David Carrera*†, Ricard Gavaldà*, Jordi Torres*† and Eduard Ayguadé*†
*Technical University of Catalonia (UPC), Barcelona, Spain
†Barcelona Supercomputing Center (BSC), Barcelona, Spain
‡Contact author. E-mail: *npoggi@ac.upc.edu*

***Abstract*—Online travel and ticket booking is one of the top E-Commerce industries. As they present a mix of products: flights, hotels, tickets, restaurants, activities and vacational packages, they rely on a wide range of technologies to support them: Javascript, AJAX, XML, B2B Web services, Caching, Search Algorithms and Affiliation; resulting in a very rich and heterogeneous workload. Moreover, visits to travel sites present a great variability depending on time of the day, season, promotions, events, and linking; creating bursty traffic, making capacity planning a challenge. It is therefore of great importance to understand how users and crawlers interact on travel sites and their effect on server resources, for devising cost effective infrastructures and improving the Quality of Service for users.**

In this paper we present a detailed workload and resource consumption characterization of the web site of a top national Online Travel Agency. Characterization is performed on server logs, including both HTTP data and resource consumption of the requests, as well as the server load status during the execution. From the dataset we characterize user sessions, their patterns and how response time is affected as load on Web servers increases. We provide a fine grain analysis by performing experiments differentiating: types of request, time of the day, products, and resource requirements for each. Results show that the workload is bursty, as expected, that exhibit different properties between day and night traffic in terms of request type mix, that user session length cover a wide range of durations, which response time grows proportionally to server load, and that response time of external data providers also increase on peak hours, amongst other results. Such results can be useful for optimizing infrastructure costs, improving QoS for users, and development of realistic workload generators for similar applications.

I. Introduction

Online Travel and ticket booking have become one of the mayor E-Commerce industries. According to the 2008 Nielsen report on Global Online Shopping [1], Airline ticket reservation represented 24% of last 3 month online shopping purchases, Hotel reservation 16%, and Event tickets 15%; combined representing 55% percent of global online sales in number of sales. Popular travel sites such as Expedia, Orbitz and Travelocity offer a mix of products including: flights, hotels, cars, cruises and vacational packages; some sites even include restaurants, activities and event tickets. Travel sites generally work as intermediaries, e.g. for airline companies by connecting themselves to Global Distribution Services (GDS) providers such as Amadeus, Galileo or Sabre, via B2B XML Web services. Depending on the product, the Travel site might also act as the provider, managing the product inventory themselves, to be used for regular users, affiliated sites, and meta-crawlers.

To offer search results, e.g. flight availability, several providers are queried and results are offered according to different algorithms of product placement in a resource intensive operation. As some of this searches are costly —not only in terms of resources— but by contract of the GDS services, meta-crawling sites such as Kayak and Travel Fusion, *scrap* travel websites simulating real user navigation in order to compare results from different sites. This situation creates the necessity to automatically identify and sometimes ban such crawlers. One commonly used strategy across the industry is to heavily rely on caching, to prevent excessive searches from meta-crawlers and speed up results for users. Moreover, visit to traffic sites might not depend only on their popularity but to current year season, holydays, search engine ranking (SEO), linking and the proximity of an event, such as a concert. The variability of the traffic creates a bursty workload, making workload characterization and modeling crucial for devising cost effective infrastructures, preventing denial of service, and improving users Quality of Service (QoS) across the application.

Evaluation of Web application resource consumption requires realistic workload simulations to obtain accurate results and conclusions. In this paper we take real production logs from a top Online Travel Agency (OTA) and make a complete characterization of both its client workload and the resource consumption, observed in the 35+ physical node cluster in which the application is deployed. The logs include several million requests over a high load week of 2010, to a 3-tier AJAX-enabled application implemented over popular LAMP[1] open-source technologies. Obtained server logs not only include HTTP data but also a detailed accounting for the resources consumed to process each request: CPU time in user mode, CPU time in system mode, number of requests and total access time to the database, time spent accessing external B2B request, as well as the current web server load status.

The characterization of the workload is approached from two different perspectives: firstly, the client workload pattern is studied, considering the request arrival rate, session arrival rate and workload pattern in a representative and generic 7 day access log. Secondly, the same 7 day log is studied from

[1]LAMP: Linux, Apache, MySQL, and PHP software

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

the point of view of resource consumption and the effect of server load on response time. The outcome of this study is the complete characterization of both user access pattern and non-simulated resource consumption of a Web application. Moreover, the studied dataset presents several features not present in most Web workload characterizations, such as the dependency of external providers, database access and mix of differentiated products (multi-application site). Resutls from this paper will support the future building of a workload generator that is able to simulate the real life characteristics of complex workloads such as the one presented here.

The rest of the paper is structured as follows: Secion II describes some of the main characterstics of the Web application (II-A) used by the OTA, their execution environment (II-B) and the datasets (II-C) made available to perform this study. Section III describes the characterization of the web workload, based in its transaction mix, intensity and burstiness. Section IV analyses the source of response time for the studied application. Section V studied how hardware resource are consumed by the application. Finally, section VI shows the related work and section VII the conclusions of our work.

II. Scenario

In this Section we provide details about the applications hosted by the OTA, its deployment scenario, and the information logged and collected for our workload characterization.

A. Applications characteristics

The application follows typical travel site structure such as the one described in Section I, offering the following products: flights, hotels, car, restaurants, activities, vacational packages, and event booking. Some of the products inventories are maintained internally e.g. *restaurant booking*, some are completely external e.g. *flights*, and some products like *hotels* are mix of internal and external providers. Access to the external providers is performed via B2B Web services. The company itself is also a B2B provider for some customers and meta-crawlers, acting as their provider via a Web Service API. The application relies on advanced caching rules, to reduce request times and load generated by the external transactions. The company's main presence and clientele is in Europe, while a small percentage of the visits are from South America, and few from the rest of the world. It is important to remark that the site is a multi-application Web site. Each product has its own independent code base and differentiated resource requirements, while sharing a common programming framework.

B. Computing Infrastructure

The studied OTAs infrastructure is composed of about 35 physical servers running GNU/Linux, connected via Gigabit switches, including: a set of redundant firewall and load-balancer servers acting as entry points and SSL decoders; about 15 dynamic web servers; 5 static content servers; 2 redundant file servers, 8 high end database servers including masters and replicas running MySQL; plus auxiliary servers for specific functions such as monitoring and administrative purposes. Web servers characterized in this study have double dual core *Intel Xeon* processors, 8G RAM and SATA hardisks. The web application runs on the latest version of PHP on Apache web servers; load is distributed using a weighted round-robin strategy by the firewalls according to each server capacity. Access to databases is balanced by using DNS round-robin rules for replica servers, most of the READ/WRITE strategy and data consistency is performed in the application itself, which also caches some queries in memory and local disc. Static content such as images and CSS is mainly served by Content Distribution Networks (CDNs), to reduce response time on the client end; the rest of the static content is served by servers running *lighttpd* and are not part of this study. In our previous work [20] we have identified that for every requested page dynamic page, about 13 static resources are accessed in average.

There are several caching mechanisms in place, for web content: there is a reverse-proxy caching static content generated by the dynamic application running *Squid*; there is a per-server caching web template caching; distributed memory key-value storage, database query cache and scheduled HTML page generators. The log file use in this study is produced by the PHP dynamic application. At the end of each executing script, a log line is generated with execution information and resource usage, detailed in the next section.

C. Dataset Properties

The main dataset used in the next experiments, consists of transactions collected over a period of one week, from Monday 03/01/2010 2AM to Monday 03/08/2010 2AM, containing 19,221,382 total dynamic requests, representing 3,269,428 distinct sessions, and 15,488 different pages (non-ambiguous URLs). The dataset is generated by the PHP dynamic application, at the end of each executing script code was added to record regular HTTP data: access time, URL, referring URL, client IP address, HTTP status code, *user agent* (type of browser), replying server. Data from the application itself: total processing time, nom-ambiguous user session id, real page requested (some URLs might be ambiguous or perform different actions), accessed product, type of request (AJAX, Administrative, etc), CPU percentage, Memory percentage and total memory, CPU time both in system and user mode, total database time, total external request time. As well as current the current server load and number of Apache processes. We have also had access to the Apache logs and monitoring system for the same week and other random weeks, these auxiliary logs have been used to validate, explain obtained results and seasonality effect. Notice that the studied dataset does not include pages that were cached by the reverse proxy or by the users browser. There is a slight cut in the dataset, where the log generator was stopped during Friday's night, but it should not affect global results.

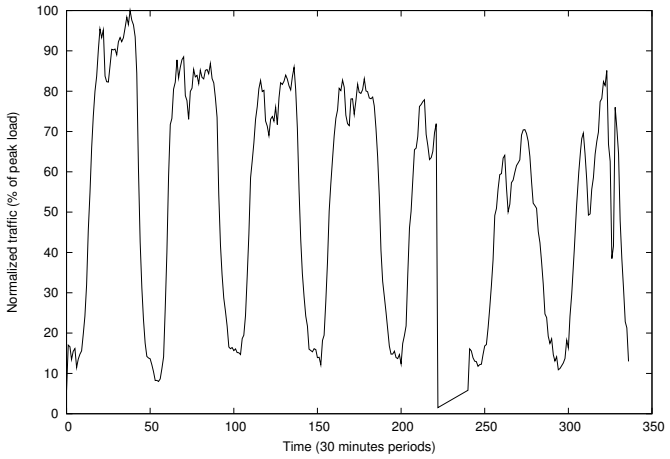


Fig. 1. Traffic volume intensity (relative to peak load). - 1 week

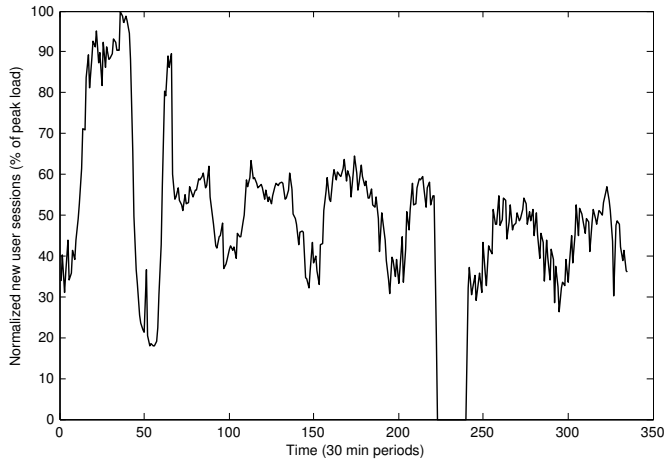


Fig. 2. New user sessions intensity (relative to peak load). - 1 week

III. Workload Characteristics and Descompositon

Figures 1 and 2 show the traffic pattern for the studied dataset, including number of hits (Figure 1) and number of user sessions started (Figure 2) over one week, grouped in 30-minute periods. Notice that data has been anonymized through being normalized to the peak load observed for each metric. As it can be observed that a problem with the logging infrastructure caused a short period of no information that can be clearly observed in Figure 2.

As it can be observed in Figure 1, the traffic decreases over the night, until it starts growing again soon after 7am in the morning. It keeps growing until noon, when it slightly decreases. Finally the workload intensity starts increasing again over the afternoon until it reaches its maximum around 9pm. Over the night, the traffic volume decreases until it finally reaches the beginning of the cycle again. Notice that client requests are conditioned by the response time delivered by the web application (next request in a user session is not issued until the response corresponding to the previous request is not received). For this reason, we made sure that the while logs were collected no severe overload conditions took place in the web infrastructure, but still capturing the a representative volume of traffic for a normal day in the online travel agency. We followed the same approach to characterize not client requests, but new web sessions in the system, that is, the number of new clients connecting to the system. The relevance of this measure, when taken in non-overloaded conditions, is that reveals the rate at which new customers enter the system. We also grouped data into 1 minute periods, that can be seen in Figure 2. As expected, per-session data follows the same trends observed for the per-request study, but with a smoother shape.

i	a	b	c
1	8.297	0.002157	1.134
2	8.072	0.002325	4.232
3	0.1572	0.009136	1.542
4	0.04958	0.01732	2.356
5	0.02486	0.02197	2.045
R-Square: 0.9701			

TABLE I
Variables of the Normalized Request Rate Function

The mean page view for the whole week is 6.0 pages per session, with 6:48 minutes spent on the site, an average of 3.0s response time for dynamic page generation, and 8MB of RAM memory consumption. Recall that the highest traffic is on Mondays and decreases to the weekend. The opposite effect is observed on average *page views* as well as the time spent on the site; they both increase during the week, peaking at the weekend, from: 5.82 and 6:40 on Mondays to 6.27 and 7:31 on Sundays, page views and time spent respectively.

$$f(x) = \sum_{i=1}^{5} a_i * sin(b_i * x + c_i) \quad (1)$$

The characterization of the normalized shape of the mean request rate for a 24h period, in 1 minute groups can be done following the Sum of Sines expression found in Equation 1, with the parameters described in Table I.

A. Workload Mix and Intensity

The workload is composed of several different request types, and for each page view that the user finally sees on his browser, several dynamic requests may have been executed. In the studied dataset we have identified the following request categories: Regular user page 46.8%, AJAX 19.8%, dynamically generated Javascript 16.6%, HTTP redirect page 9.1%, Administrative 4.5%, internal scheduled batch 3.1%, API Web Service 0.04%, and Extranet 0.02%. It is an important feature of this dataset (and probably other real-life logs) that less than 50% of the total dynamic requests correspond to user clicks on their browsers.

Figure 3 shows the fraction of dynamic traffic volume that corresponds to different types of request categories, focusing on most relevant ones: Regular, AJAX, Redirects and JavaScript contents. As it can be observed, AJAX content

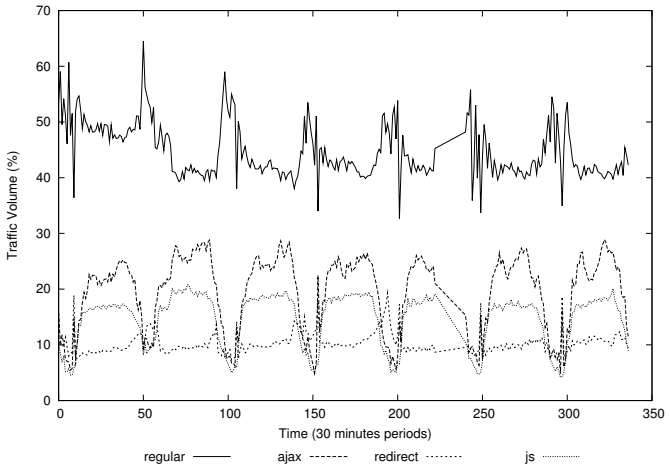


Fig. 3. Traffic mix over time (transaction type) - 1 week

Application	Percentage
App 1	27%
App 2	17%
App 3	15%
App 5	6%
App 6	5%
Other	23%

TABLE II
PERCENTAGE OF REQUESTS PER APPLICATION

fraction is mainly correlated to site's load, as such traffic is usually generated by human-generated actions (e.g. auto-completion of forms when indicating flight origin and destination airports during a search). During low traffic periods, basically overnight, most of the traffic is identified as regular pages. Night traffic also involved most of the internal batch and crawler activities.

A brief analysis on the number of crawler requests and meta-crawlers by analyzing the *agent* field (the reported browser type) identifying themselves as such; our findings indicate that the number of *bot* requests is about 15% of the total traffic. This is consistent with previous work on a similar dataset 3 years before [19], that identified between 10% and 15% total bot content. Even more traffic may correspond to crawlers and meta-crawlers assuming that some might simulate being a real user when accessing the site, that would show a growing trend in the proportion of automated bot traffic.

Table II shows traffic distribution across anonymized products (applications) offered by the OTA described in II-A. As it can be observed, almost 60% of the overall traffic come from only three applications, representing the most popular products of the company. Although each application is implemented independently, they share a common code base (e.g. user logging and shopping cart). Such *common* activity is not included in the specific per-application traffic volume, and is considered as a separate application by itself, corresponding to App 3, 15% of the total requests; this distribution is site specific.

Next step in the workload characterization is to study the per-session characteristics of the OTA visitors. Each session is started when a new visitor comes into the system, and is identified through a single identifier in the workload trace. We will look at four different session-specific characteristics: number of different products visited during the session, number of different pages visited per session, number of hits per session (notice that a hit can be initiated by a user click or by Javascript events such as auto-complete controls), and the session length. For each one of these characteristics, we construct a CDF chart as shown in Figure 4. Each CDF is built from the information collected during the lifetime of all the sessions started within a 30 minutes period. Recall that the completion time of a session can be much later than the end of the 30 minutes period. We have explored 4 different time ranges for each property, selecting time ranges corresponding to 4 points of time with different traffic characteristics, including night, morning, afternoon and evening traffic. The selected time ranges are 5:00am to 5:30am, 11am to 11:30am, 4:00pm to 4:30pm, and 10:00pm to 10:30pm. It can be seen from the Figures that all properties remain unchanged for all time ranges except for the night one. Session characteristics are approximately the same for morning, afternoon and evening time ranges, but a clear difference can be seen for the night (5am) traffic. Notice that the OTA is international, most of the traffic come from European countries located within the time zones with a maximum of 2h of difference. Obviously, the different characteristics of the nightly traffic come from the fact that the many sessions are initiated by non-human visitors (bots), including crawlers and meta-crawlers. This result supports the results presented before in Figure 3. Daytime (10pm) CDFs can be approximated using the probability distributions and parameters listed in Table III.

Our study concluded that 75.03% of the sessions only contained 1 hit, that is, the user only accessed 1 page, and then just quit the OTA site. This is mainly due to many visitors reaching the site through external banners that redirect them to espcial *landing pages*, and many of these users do not continue browsing the OTA site after this initial page. In the building of the CDFs, 1-click sessions were excluded as we want to study customer's characteristics; 1-click sessions are included in the rest of the experiments.

Figure 4(a) shows number of different pages visited per session (notice that a page is a unique URL here). Most users visit few pages during a session, and they may remain in one single page running searches or browsing the OTA catalog. Some users visit up to 14 pages in one single session, but that is the least of them. Figure 4(b) shows number of hits per session, with half of the visitors producing 10 or less requests to the OTA site. Notice that a request can be initiated by a user click, or by an AJAX action, such as field auto-completion in search forms. A significant percentage of visitors produce many more requests, reaching a few tenths in many cases. Figure 4(c) shows number of products visited per session. As the OTA site contains several different products, each one associated to a particular web application, we were interested

in studying how many different products were visited by each individual session. It can be seen that around 50% of the customers are interested in only two different products, but in some cases 8 or even more products may be visited in one single session. Finally, Figure 4(d) shows session length CDF, showing that while most visitors sessions last only a few minutes, some of them may be active for several hours. That may be explained by users coming back to the OTA site quite often over a long period of time, or by the presence of crawlers that periodically poll the OTA site contents.

Finally, we look at the burstiness properties of the workload, paying special attention to the session arrival rate and its changes over time. For such purpose, we have characterized the Index of Dispersion for Counts (IDC) of the entire workload as well as for a shorter time period which presents stationary properties. The IDC was used for arrival process characterization in [11], and has been leveraged to reproduce burstiness properties in workload generators in [5]. IDC was calculated by counting sessions started in 1 minute periods In a first step, we characterized the IDC for session arrival rate for the full dataset, covering one week period. The result for this step is shown in Figure 5(a). In a second step we picked the stationary period shown in Figure 5(c), corresponding to a 500 minutes high-load period, and characterized its burstiness through its IDC, as shown in Figure 5(b). Both figures indicate, given the high value of IDC observed, that the workload shows a high degree of burstiness as it is expected for any web workloads. And it remains true at both scales, including one week of processed data and a short and clearly stationary period of logged data.

IV. Response Time Analysis

In the following section we perform an analysis on response time: how it varies during the day, how it is affected by server load, how it affects the different applications, and finally it effects on *user behavior*. Total response time for a request is the time it takes Web servers to start sending the reply over the network to the user's browser. It is important to notice that in the OTA's application, *output buffering* is turned *on* for all requests, so no data is sent over the network until the full request is processed and *gzipped*, if supported by the user's browser. There is an additional time for the browser to render the final webpage, but it is not present in our dataset and is not part of this study as it deals with the actual HTML and media content.

A. Response Time Decomposition

From the available dataset, response time can be decomposed into: CPU time in system mode, CPU in user mode (including I/O times), database wait time, and external request wait time. Figure 6 presents the total response time for the complete dataset grouped by hour of the day. If we contrast it with Figure 1, by each daily period it can be seen clearly that response time increases with the total number of requests. Figure 6 also divides total time by the different resources, where the database response time also increases at peak hours. External request response time is not affected in the same proportion. CPU in system mode is not plotted on the graph as it was too low in comparison to the rest of the features; however it also presented noticeable higher response times at peak load. At peak time, from 18 to 22hrs, as Web server process more requests, they also present some resource contention due to high *load average* detailed in the next section.

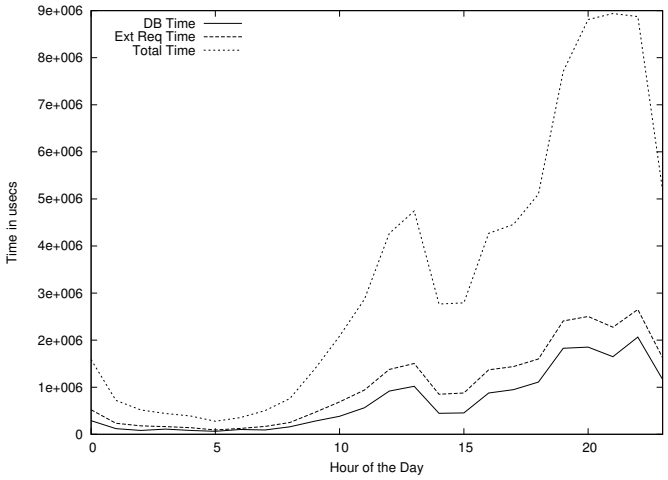


Fig. 6. Variation of response time during day from 0 to 24hrs

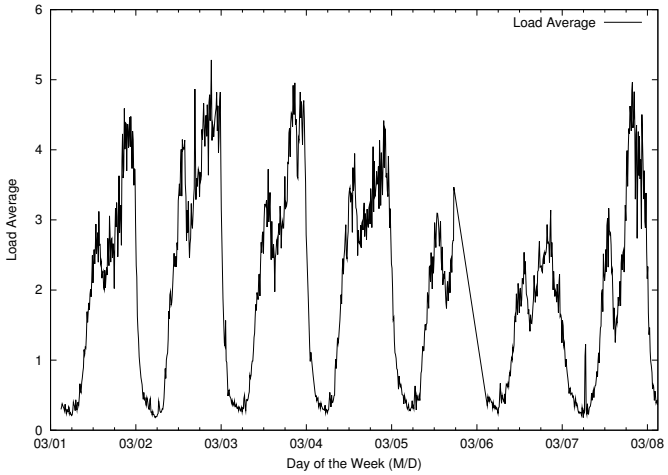
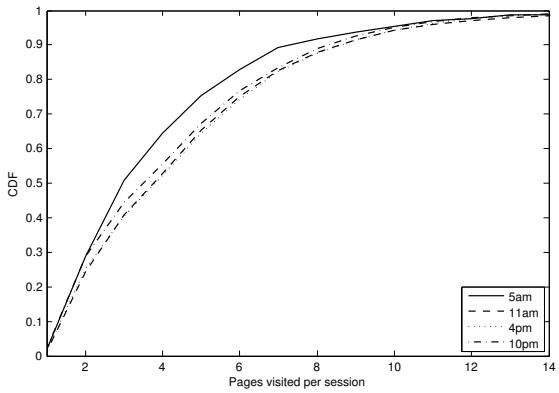


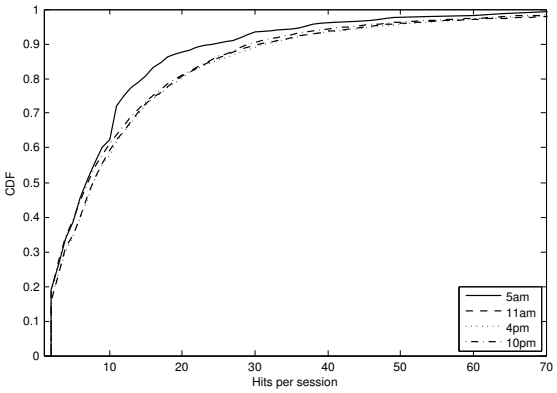
Fig. 7. Load Average values for the week

B. Response Time and Server Load

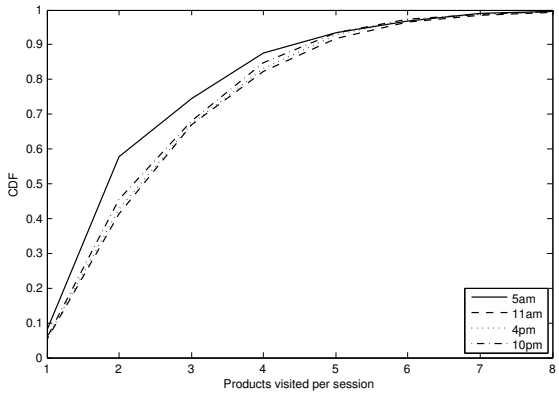
The next section analyzes how response time is affected as the *load* on the Web servers increases. To measure server load, we take the *load average* system value present in most UNIX systems [10], recall that the value of the *load average* is related to the number of scheduled jobs pending to be processed by the CPU. *Load average* is a very extended, simple, but accurate value for measuring current load on a server; in this study we use load averaged to 1 minute —opposed to 5 or 15 minutes— to have higher detail. To understand how loaded is a server by the load average, it is important to notice that each Web server has 2 CPUs with 2 cores each (described in II-B), giving a



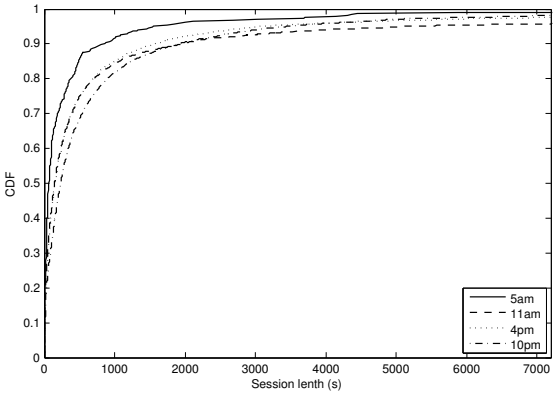
(a) Pages per session CDF



(b) Hits per session CDF



(c) Products per session CDF

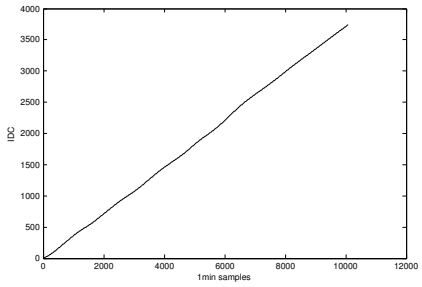


(d) Session length CDF

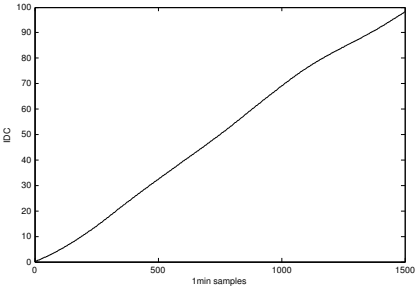
Fig. 4. Session properties - grouped according to session start time in 30-minute bins

Pages per Session	
Log-normal	$\mu = 1.37536; \sigma = 0.60544$
Hits per Session	
Log-normal	$\mu = 2.05272; \sigma = 1.00659$
Products per Session	
Log-normal	$\mu = 1.01541; \sigma = 0.457558$

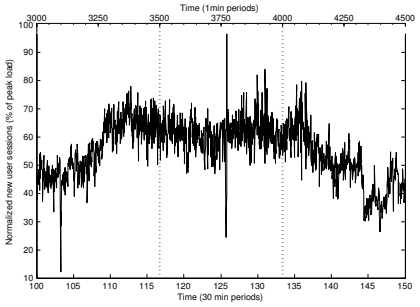
TABLE III
PER SESSION CDF FITS



(a) 1 week period - 1 min bins



(b) Stationary period - 1500 minutes - 1 min bins



(c) New user sessions intensity (relative to peak load). - 1 week

Fig. 5. Workload Burstiness: Index of Dispersion for Counts (IDC) of Initiated User Sessions

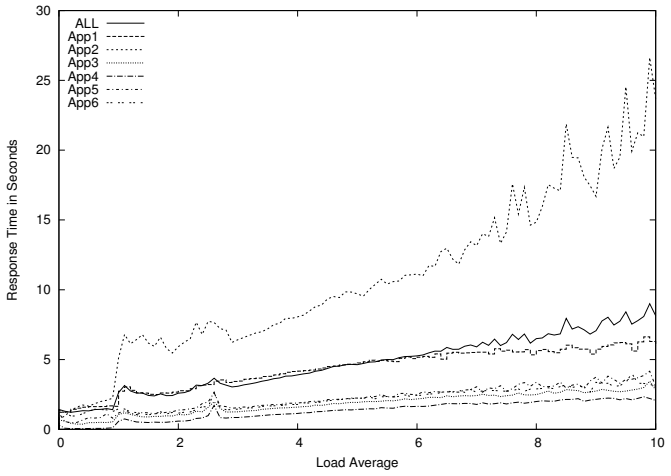


Fig. 8. Response Time by Load for different Apps

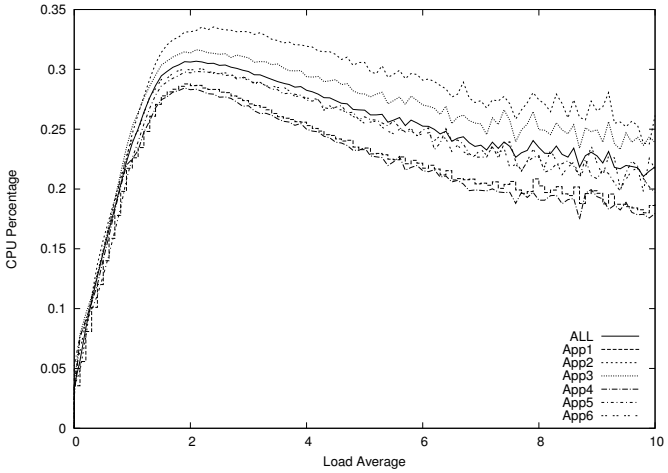


Fig. 9. Percentage of CPU Assignation by Load for Different Apps

total of 4 cores per server; as a rule of thumb, after 4 units of *load average* servers are considered overloaded (1 for each core).

Figure 7 presents the *load average* of the servers during the one week dataset. If we compare Figures 1 and 7 we can correlate how *load average* is affected directly by the number of concurrent requests on a given time, and that it follows the daily request pattern.

In Figure 8 we plot response time (Y axis) and *load average* (X axis) for the most popular applications, Apps 1 through 6 and the average *ALL* for all applications. Load average starts from 0, being the least busy value, to 10, the maximum value recorded on our dataset. From Figure 8 it can be appreciated that response time increases almost linearly as server load increases. From load 0 to 10, it increases almost to 10x in *ALL* for all applications, and up to 25x for App 2.

Response time increases with server load for three main reasons: server resource starvation (less dedicated system resources for a request), external B2B requests increased response time (low QoS), and contention between resources (jobs waiting for blocked resources).

For server resource starvation, Figure 9 shows how the percentage of CPU assigned by the OS to a specific Apache thread (request) reaches a maximum at load 2 (saturation point), and then starts decreasing leading to higher response time values. The same effect happens with the percentage of assigned memory, Figure 10, plots how memory percentage to Apache threads decreases very steeply from load 2.

As for external resource QoS, Figures 11 and 12 shows the response time for database queries and external B2B requests respectively. In Figure 11 we can see how the database response time also increases 3x in average for all applications, and up to 8x for App 2, which has a higher database usage. Figure 12 shows the average response time to external requests, we can see that App 2 is affected highly by the QoS of the external provider(s), while for App 1 it stays constant. The effect on App 2 is caused by the external providers getting overloaded at similar day times, than the

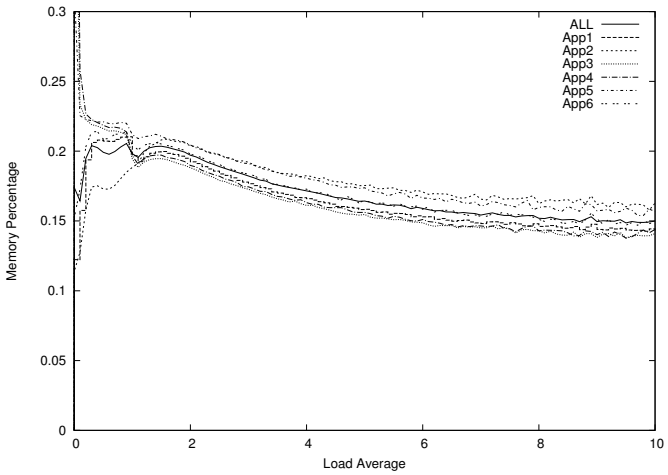


Fig. 10. Percentage of Memory Assignation by Load for Different Apps

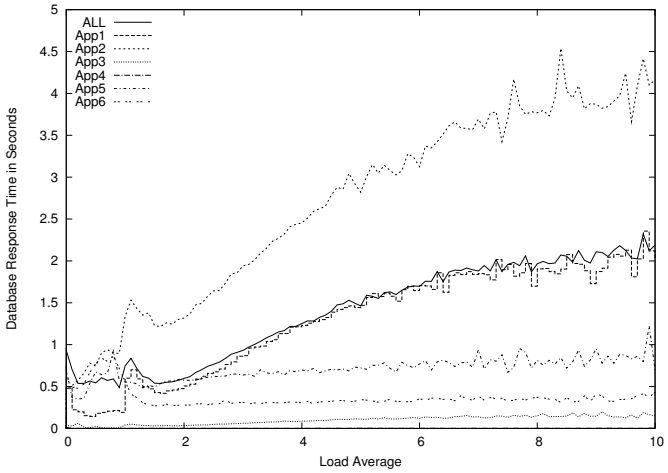


Fig. 11. Database Response Time by Load for Different Applications

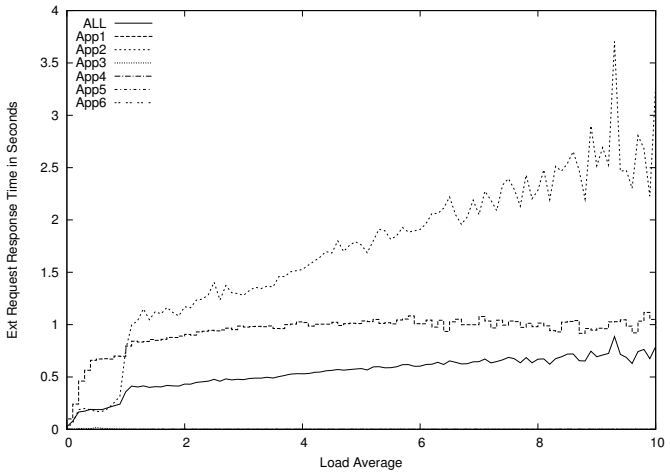


Fig. 12. External Request Time by Load for Different Applications

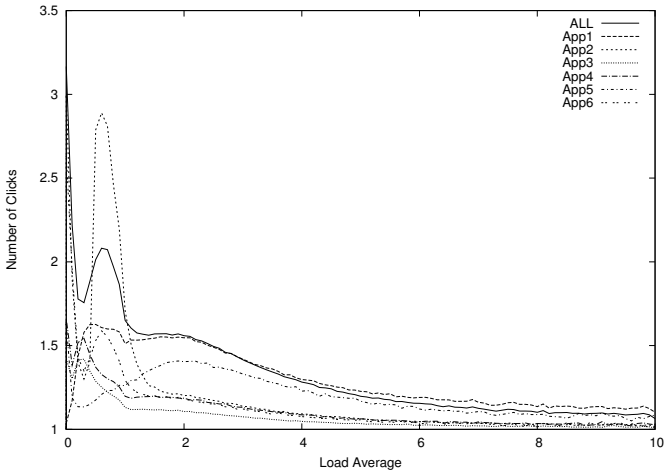


Fig. 13. Number of Clicks by Load

analyzed site; while the QoS for App 1 is stable at the sites peak hours.

It is important to notice that the less affected application by server load, is App 3, it is clear from Figures 11 and 12 that it does not rely on database or external requests, having the lowest response time increase, 2.5x. The other extreme, App 2, is heavily affected by both the database and external request times. An important feature from the last figures, if we zoom into the graph, is that the best response time is not at load 0, but is between load 0 and 1, as at load 0 (mainly at night time) *cache* hit rate is lower which leads to slightly higher times, although not comparable to high *load average*.

C. Sever load effect on Users

In the previous subsection, we have established how *response time* increases as load on servers increases following a linear trend. *Response time* has a direct impact on user behavior on the site, Figure 13 shows how the average number of clicks decreases as load and consequently, response time increases. This is consistent with previous works on user behavior [7], [21].

V. Resource Consumption

Figure 14 shows resource consumption distribution across anonymized applications.

When modeling sessions and requests they also have different characteristics and resource requirements. Figure 15 shows the different resource percentage used by each type of requests.

In Figure 16 we pick the most popular product of the OTA company and characterize the interaction of its code with both the database tier and external providers of information. The characterization is done by building the CDF of each metric, what can be approximated using the functions seen in Table IV.

All the services related to this product require accessing at least once at the DB tier. Figures 16(a) and 16(b) show the CDF of the number of DB queries per request and the time spent per request waiting for the result of DB queries.

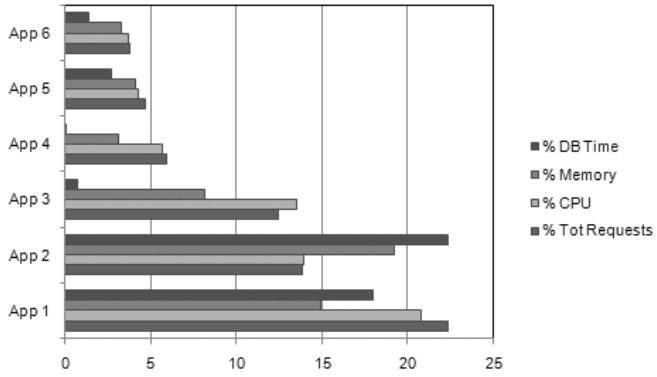


Fig. 14. Percentage of Resources by Application

Recall that this information corresponds only to the most popular product of the OTA. As it can be observed, 50% of the requests issue 1 or 2 queries to the DB tier, and around 80% of the requests require less than 10 queries to complete. But a significant fraction of the requests produce complex results and require a large number of DB queries to be completed, reaching more than one hundred DB requests in some cases. Looking at the time spent waiting for data from the DB tier,

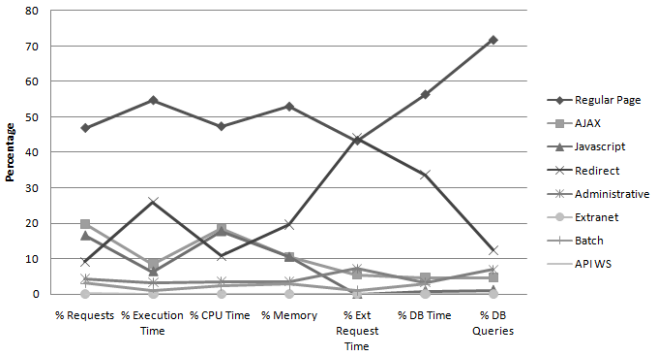
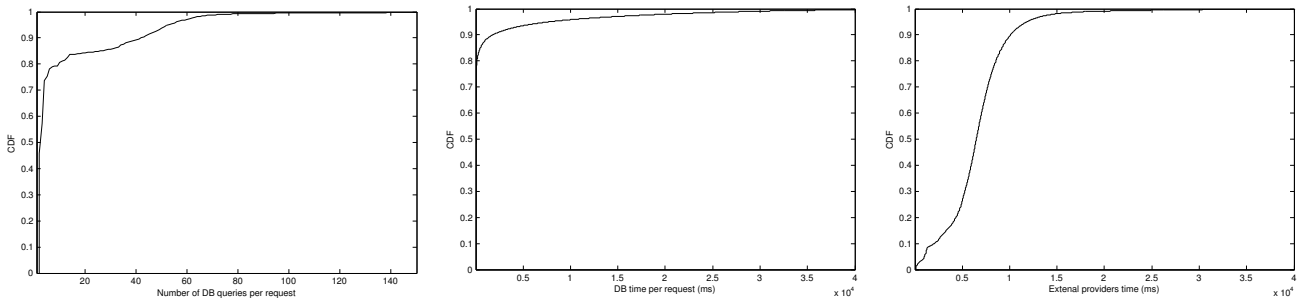


Fig. 15. Percentage of resource usage by request type



(a) DB queries (b) DB time (c) External Providers Time

Fig. 16. CDF of resource consumption for most popular product during an stationary, high load, 500 minutes period

Metric	Model	Parameters
DB Time	Weibull	$a = 30141.4; b = 0.251286$
DB Queries	Generalized Pareto	$k = 0.61979; \sigma = 4.65092; \mu = -0.1$
Ext. Provider Time	Logistic	$\mu = 6.17049e + 07, \sigma = -191940$

TABLE IV
DB AND EXTERNAL TIME CDF FITS FOR MOST POPULAR PRODUCT

most of the requests exhibit just a couple of seconds of DB query waiting time, but some cases can reach up to nearly 40s of DB time. Notice that some OTA operations may require complex optimization operations, as well as may provide a long list of results based on user search parameters.

Looking at the interaction between the OTA and external information providers, it has been observed that the probability of accessing an external provider follows a Binomial distribution with parameters $n = 1,125,969; p = 0.1306$ for App1. For those requests that did involve access to an external site, Figure 16(c) show the CDF of the time spent waiting and processing the information provided by the external source. As it can be derived from this information, caching techniques are effectively used for this application, avoiding in many cases (more than 75%) the cost of gathering information from external providers. For the cases in which accessing an external provider is required, the process is usually completed in less than 20s.

VI. RELATED WORK

In the context of Web workload analysis, there are few published studies based on real e-commerce data, mainly because companies consider HTTP logs as sensitive data. Moreover, most works are based on static content sites, where the studied factors were mainly: file size distributions, which tend to follow a Pareto distribution [2]; and file popularity following Zipfs Law [2], [12], [24]. Also, works such as [13] have studied logs from real and simulated auction sites and bookstores; there are no studies that we know about which are concerned with travel sites, like the one studied here, where most of the information comes from B2B providers and have a different behavior. Other works come to the same conclusions, but from the point of view of generating representative workload generators, such as [3] for static workloads and [6] and [16] for dynamic applications. None of these studies looked in detail at a complex multi-product Web 2.0 application in production of the scale of what is studied in this paper.

Similar work was conducted in [8] but following a black box approach for the enterprise applications, not exclusively web workloads, meaning that they shown no information about the nature and composition of the studied applications. Their work was data-center oriented, while our work is application-centric.

Recent studies have performed similar workload characterizations as the one presented here. In [4] Benevenuto et al. characterizes user behavior in *Online Social Networks* and Duarte et al. in [9] characterizes traffic in Web *blogs*. Previous work on the characterization of collaborative web applications was conducted in [23]. Although both the *blogosphere* and the OTA application used in our work are similar in the sense that they are user-oriented, user behavior is different in these scenarios. Moreover a more detailed analysis is presented in this paper, as the site is multi-application, and applications are further subdivided to perform a separate analysis by day, type of request, applications, as well as the resource consumption by each.

Few studies present both a characterization of workload and resource consumption. In [17] Patwardhan et al. perform a CPU Usage breakdown of popular Web benchmarks with emphasis on networking overhead, identifying that network overhead for dynamic applications is negligible, while not for static content. In [25] Ye and Cheng present a similar characterization of resource utilizations as the one presented here, but for *Online Multiplayer Games*. In this paper we also cover how response time affects user behavior in session length and number of clicks, validating results from previous studies [7], [21].

While [11], [5], [14] and [15] discuss about the need of stationarity of arrival processes to study and characterize workload burstiness, in [22] the authors work on non-stationary properties of the workloads to improve performance prediction. In our work, we have leveraged the techniques presented in some of these studies to characterize workload burstiness in stationary periods, but have not extended this work.

VII. Conclusions

In this paper we have presented a workload and resource consumption characterization for an Online Travel Agency. We were given access to a large dataset including millions of records for over one week of web activity, including workload information as well as delivered response time, and resource consumption levels observed on the underlying infrastructure. The online site is developed following modern technologies commonly used in the Web 2.0. In our work we decomposed the workload to create a clear picture of products (applications), unique pages, request types, system resources, interactions with databases, and external web services such as Global Distribution Services (GDS).

Results have been grouped into three categories: workload characterization, including transaction mix, intensity and burstiness; response time decomposition, showing sources of delay and effects of server load on response time and user clicks; and resource consumption, distinguishing between applications, putting emphasis to databases and external providers. Results show that the workload is bursty, as expected, that exhibit different properties between day and night traffic in terms of request type mix, that user session length cover a wide range of durations, that response time grows proportionally to server load, that response time of external data providers also increase on peak hours, and that automated crawler traffic is increasing and can represent more than 15% of total traffic, amongst other results.

As future work we plan to build a synthetic workload generator that mimics the studied application to model resource usage and bottlenecks. We also plan to analyze further, how response time affects user satisfaction and what would be the optimal server configuration to improve user QoS, while minimizing server costs by using techniques such as dynamic server provisioning [18].

Acknowledgements

This work is partially supported by the Ministry of Science and Technology of Spain and the European Union under contracts TIN2007-60625, TIN-2008-06582-C03-01, and by the Generalitat de Catalunya (2009-SGR-980).

References

[1] Trends in online shopping, a Nielsen Consumer report. Technical report, Nielsen, Feb. 2008.

[2] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In *DIS '96: Proceedings of the fourth international conference on on Parallel and distributed information systems*, pages 92–107, Washington, DC, USA, 1996. IEEE Computer Society.

[3] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. *SIGMETRICS Perform. Eval. Rev.*, 26(1):151–160, 1998.

[4] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *IMC '09: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 49–62, New York, NY, USA, 2009. ACM.

[5] G. Casale, N. Mi, L. Cherkasova, and E. Smirni. How to parameterize models with bursty workloads. *SIGMETRICS Perform. Eval. Rev.*, 36(2):38–44, 2008.

[6] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. *SIGPLAN Not.*, 37(11):246–261, 2002.

[7] D. F. Galletta, R. Henry, S. Mccoy, and P. Polak. Web site delays: How tolerant are users. *Journal of the Association for Information Systems*, 5:1–28, 2004.

[8] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. pages 171–180, Sept. 2007.

[9] M. A. Goncalves, J. M. Almeida, L. G. dos Santos, A. H. Laender, and V. Almeida. On popularity in the blogosphere. *IEEE Internet Computing*, 14:42–49, 2010.

[10] N. J. Gunther. Performance and scalability models for a hypergrowth e-commerce web site. In *Performance Engineering, State of the Art and Current Trends*, pages 267–282, London, UK, 2001. Springer-Verlag.

[11] R. Gusella. Characterizing the variability of arrival processes with indexes of dispersion. *Selected Areas in Communications, IEEE Journal on*, 9(2):203 –211, feb 1991.

[12] M. Levene, J. Borges, and G. Loizou. Zipf's law for web surfers. *Knowl. Inf. Syst.*, 3(1):120–129, 2001.

[13] D. Menascé, V. Almeida, R. Riedi, F. Ribeiro, R. Fonseca, and W. Meira, Jr. In search of invariants for e-business workloads. In *EC '00: Proceedings of the 2nd ACM conference on Electronic commerce*, pages 56–65, New York, NY, USA, 2000. ACM.

[14] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Burstiness in multi-tier applications: symptoms, causes, and new models. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 265–286, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[15] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In *ICAC '09: Proceedings of the 6th international conference on Autonomic computing*, pages 149–158, New York, NY, USA, 2009. ACM.

[16] P. Nagpurkar, W. Horn, U. Gopalakrishnan, N. Dubey, J. Jann, and P. Pattnaik. Workload characterization of selected jee-based web 2.0 applications. pages 109–118, Sept. 2008.

[17] J. P. Patwardhan, A. R. Lebeck, and D. J. Sorin. Communication breakdown: analyzing cpu usage in commercial web workloads. In *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 12–19, Washington, DC, USA, 2004. IEEE Computer Society.

[18] N. Poggi, T. Moreno, J. L. Berral, R. Gavald, and J. Torres. Self-adaptive utility-based web session management. *Computer Networks Journal*, 53(10):1712–1721, 2009.

[19] N. Poggi, T. Moreno, J. L. Berral, R. Gavald, and J. Torres. Automatic detection and banning of content stealing bots for e-commerce. *NIPS 2007 Workshop on Machine Learning in Adversarial Environments for Computer Security*, December 8, 2007.

[20] N. Poggi, T. Moreno, J. L. Berral, R. Gavald, and J. Torres. Web customer modeling for automated session prioritization on high traffic sites. *Proceedings of the 11th International Conference on User Modeling*, pages 450–454, June 25-29, 2007.

[21] P. J. Sevcik. Understanding how users view application performance. *Business Communications Review*, 32(7):8–9, 2002.

[22] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. *SIGOPS Oper. Syst. Rev.*, 41(3):31–44, 2007.

[23] C. Stewart, M. Leventi, and K. Shen. Empirical examination of a collaborative web application. pages 90–96, Sept. 2008.

[24] A. Williams, M. Arlitt, C. Williamson, and K. Barker. Web workload characterization: Ten years later. *Springer*, 2005.

[25] M. Ye and L. Cheng. System-performance modeling for massively multiplayer online role-playing games. *IBM Syst. J.*, 45(1):45–58, 2006.

Characterizing Datasets for Data Deduplication in Backup Applications

Nohhyun Park* and David J. Lilja†
Department of Electrical and Computer Engineering
University of Minnesota, Minneapolis, MN 55455
{*parkx408, †lilja}@umn.edu

***Abstract*—The compression and throughput performance of data deduplication system is directly affected by the input dataset. We propose two sets of evaluation metrics, and the means to extract those metrics, for deduplication systems. The First set of metrics represents how the composition of segments changes within the deduplication system over five full backups. This in turn allows more insights into how the compression ratio will change as data accumulate. The second set of metrics represents index table fragmentation caused by duplicate elimination and the arrival rate at the underlying storage system. We show that, while shorter sequences of unique data may be bad for index caching, they provide a more uniform arrival rate which improves the overall throughput. Finally, we compute the metrics derived from the datasets under evaluation and show how the datasets perform with different metrics.Our evaluation shows that backup datasets typically exhibit patterns in how they change over time and that these patterns are quantifiable in terms of how they affect the deduplication process. This quantification allows us to: 1) decide whether deduplication is applicable, 2) provision resources, 3) tune the data deduplication parameters and 4) potentially decide which portion of the dataset is best suited for deduplication.**

I. INTRODUCTION

Data deduplication provides a means to efficiently remove redundancies from large datasets. This process is made possible by first dividing up the data into segments and representing each segment with a much smaller hash value. A redundant segment of data is then easily identified through a hash table lookup. The efficiency of the process comes at the cost of possible data loss through hash collisions, thoug it is understood that the collision probability is negligible compared to the soft error rate of the storage system if an appropriate hash function is used [1]–[4]. Another performance factor is the granularity of compression which is limited to the size of duplicate segments. Two segments that are off by a single bit will result in no compression.

Despite these costs, data deduplication has steadily gained its place in backup [2], [5], [6], archive [7] and virtual machine storage solutions [8]–[10] due to its potentially huge reduction in storage space and IO elimination. However, as more and more systems opt to take advantage of data deduplication techniques, the variability in performance is becoming an issue. The cause of this variation can be catagorized into two factors, *systemic variation* and *input variation*. Systemic variation is caused by the use of different algorithms and techniques as well as the underlying hardware deployed in the deduplication systems. The input variation is caused by different characteristics of the input datasets. The systemic variation is critical from the deduplication system designers' and vendors' perspectives since it allows them to compare two systems directly. However, the input variation is typically more critical from the custumer's perspective when evaluating the potential benefits of data deduplication for different types of datasets.

Current metrics for characterizing the datasets for data deduplication are overly simplified and often inaccurate. For example, the compression ratio (CR) is typically estimated using the *average data change rate* ($\overline{dcr}$), which is the percentage of data change. Let R be the required retention period. Assuming a full backup per unit time, the compression ratio is simply:

$$CR = \frac{\text{Compressed Size}}{\text{Uncompressed Size}} = \frac{1 + (R-1) \cdot \overline{dcr}}{R}. \quad (1)$$

This model provides a simple estimation of the ompression ratio but it can also be very inaccuarate. We observed over 35% error using the $\overline{dcr}$ to charaterize one of our datasets. One of our main contributions is providing a new set of metrics and models that allow much more accurate compression performance predictions to be made. In five out of six cases we test, the error reduction was over 50% when compared with the $\overline{dcr}$ method.

The main performance metrics for data deduplication systems are the compression ratio and the read/write throughput. While the latency could also be an issue for virtual machines, it can be mostly masked by high level caching and prefetching mechanisms. Unless specified otherwise, we use the term *performance* to represent both the throughput and compression together.

The throughput of the system is heavily dependent on both system and input factors. Thus, it is difficult to determine if one dataset outperforms the other in terms of throughput. However, there are features of datasets that are likely to benefit throughput in most systems. One obvious one is the

This work was supported in part by National Science Foundation grant no. CCF-0621462, the Center for Research in Intelligent Storage (CRIS), which is supported by National Science Foundation grant no. IIP-0934396 and member companies, and the Minnesota Supercomputing Institute.

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

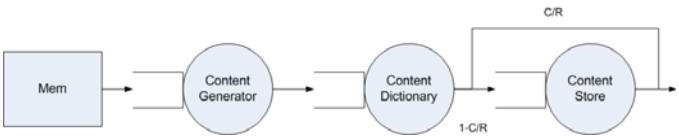


Fig. 1. Our simplified deduplication system model. It is essentially 3 server open queuing system. At the content dictionary, C/R of the segments exit the system and 1-C/R of the segments are passed on to content store. All data are assumed to be immediately available at the memory and all three servers have exponentially distributed service time.

compression ratio itself. A highly compressable dataset has lower IO requirements which in turn improves the throughput in most systems. The compression ratio is the most influential factor in determining the throughput. However, there are also other factors, such as spatial locality of the data segments and bursty arrival rates, which we examine in this paper.

Our main contribution in this paper is to provide a framework to characterize the input dataset for data deduplication systems. We 1) show that different datasets behave with a unique pattern that is quantifiable. Furthermore, we 2) provide analysis on how this pattern affect the deduplication system performance. As part of the framework, we 3) provide classification of segment types and their relations. We further provide parameters to show how the *composition* of these segment types change over time.

Section II will describe the data deduplication more formally and define set of essential characteristics of data deduplication systems. In Section III, we describe the datasets and the main attributes for which we characterize of the data. Section IV describes our analysis and Section V propose various applications of those results. Section VI lists related works, Section VII lists our limitations and Section VIII concludes the paper.

II. System Components

The deduplication process itself can be divided into largely three parts as shown in Fig.1. First part of the process generates a sequence of segments from input data stream which we call the *content generator*. In the second phase, the deduplication system generates a dictionary of segments which is in turn used to identify the duplicate segments which we call the *content dictionary*. In the last phase, new segments are stored on a persistent storage system which we call the *content store*. The later two components form a *content addressable storage*(CAS).

The input data itself is another system component that affects the performance. To understand how the dataset affects the performance, it is described in terms of how the data components can be separated once it is deduplicated. The segments, regardless of how they are defined, are the smallest units of data in data deduplication. therefore, we define data as a sequence of segments rather than contiguous bits or characters. In this work, we show that the distribution of these different segment types completely describes how the dataset affect the deduplication performance.

A. Content Generator

The goal of the content generator is to maximize the redundant segments while minimizing the number of segments generated. These two goals often conflict with each other since the amount of duplicate data tends to increase with smaller segments size. This is due to the fact that the generation of segments are typically done in stateless manner to keep up with the huge data size and stringent throughput requirements. Recent papers try to overcome this limitation through a two level definition of contents [3], [11]. However, these approaches come at the extra computation cost and are applicable only when there are less stringent throughput requirements.

These are the three most common methods to define the segment boundaries also known as the *anchors* [2], [4].

- *File based*: A file boundary becomes the segment boundary.
- *Size based*: The anchors are designated every k bits which determines the boundaries for the segment. Therefore, any addition or deletion of data results in shifted anchors which is propagated until the end of the data stream. This effect is known as the *boundary shifting problem* [4].
- *Content based*: An anchor is generated based on the content of the data which becomes the boundaries for the segments. Therefore the anchors are shifted together with the contents in the case of addition and deletion of the data.

The *content based* approach is the most popular method used in data deduplication due to the boundary shifting problem found in the size based method. Average, maximum and/or minimum segment size are usually specified for the content based approach. This allows system to expect segments of the bounded size which makes the segment handling simpler but also ensures that amount of size variation is limited which can cause loss of redundancy [12].

From the system perspective, the content generator is the source generator for the CAS. Typically, any information about the original data stream is lost after this point. This justifies our looking the dataset as a sequence of segments since beyond this point, the system is effectively decoupled from the original dataset.

B. Content Dictionary

The content dictionary is typically a hash table which is keyed by hash of segments. Hash functions used typically provide one-way property as well as collision resistance. While only the collision resistance is required for correct functionality, one-way property is also attractive to ensure that the malicious users cannot corrupt the system by generating hash collisions.

For a large dataset, the content store itself becomes significantly large. To relax the memory requirements for the deduplication system, the content dictionary is typically stored on the disk [2], [6], [13], [14] . Only a portion is cached onto the memory at a time. Furthermore, any writes to the index

table must be serialized which require expensive locks [10]. Therefore, the caching algorithm for the index table is one of the major performance factor within the deduplication system.

To avoid unnecessary accesses to the index table, Bloom filter [15] is used to quickly determine segments that are not in the index table [2]. While the backup stream typically exhibit an inherent spatial locality between the segment instances [2], more intelligent caching schemes that uses data similarities have also been proposed [6], [14].

Regardless of caching scheme, it is clear that content dictionary determines the data path for any given segment. While every segment must read the index table at least once, only the new segments result in it's update*. Furthermore, these new segments must be passed down to the content store to be store on the disk. Problem is more complicated for the read where segments maybe fragmented in various places both for the actual contents and the dictionary. A recent work duplicated heavily used segments over the physical disk such that the average seek time can be minimized [16]. However, such approach assumes small working set of segments and is heavily workload dependent.

In this work we assume nothing about the caching policy of the content dictionary or any other auxiliary data structure to minimize the access to the dictionary. We only assume that the different segment types results in different resource requirements in regular read/write operations.

C. Content Store

Once the segments are identified as new content that needs to be stored, it is passed down to the content store to be processed. Various different mechanisms are possible. The content may already be on the disk and the content store only generates a logical mapping and freeing of segments [10], [17] or use log-based filesystem to optimize for the in-band writes [2].

The content store is not affected specifically by the deduplication system. While inherent *copy-on-write* (COW) support of storage subsystem [18], [19] allows easier sharing of segments, the fundamental operations of underlying storage does not differ from dataset to dataset. Therefore, we focus mostly on the behaviors of the content dictionary in this paper.

D. Queuing Model

The system can be viewed as simple open queuing system with three servers as seen in Fig.1. We make very little assumptions of particular algorithms deployed. However, the functionality of each stage we have just described exists in all conventional deduplication systems. In this system, the arrival rate of segments at the *content dictionary* is assumed to be exponentially distributed with the mean of average service rate at the *content generator*. The probability of the arriving segments to be found in the dictionary is equal to the the compression ratio in which case it can by pass the *content store*. If we assume that each outcome is independent, the arrival rate at content store is also exponentially distributed. However, we show that this is not true and is one of the factors that affect the throughput of the system.

*This may not be strictly true if there exists a metadata information which must be updated within index table for duplicate segments as well. However, the fact that the new segments incur extra operations does not change.

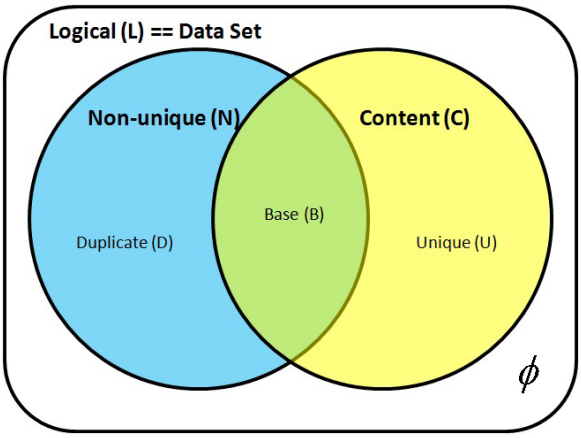


Fig. 2. Venn Diagram of Segment Sets. The *logical segments* which is equivalent to the *dataset*, can be grouped into two separate sets N and C. The elements of N are segments whose content occurs more than once in the dataset while the elements of C are segments that must be stored in order to ensure no content information is lost from the dataset.

E. Segment Classification

We first define data segment as a tuple of *offset* and *size* within a dataset. Therefore every segment is uniquely identifiable regardless of its content. The only constraint under this definition is that the size of the data segment must be less than or equal to the size of the dataset. Also, the content of the data segment must exist within the dataset. There are no other restrictions to what may be defined as a segment. A dataset is then a mere concatenated sequence of data segments.

We begin classifying the data segments by defining the *logical segments* which is the set of all segments from which you can reconstruct the original dataset through reordering only. Under this definition, a *dataset* is nothing but an ordered list of logical segments. We also define *unique segments* and *non-unique segments*. The unique segments are a set of segments whose content is unique across the logical segments. Obviously the non-unique segments are the segments whose content is repetitive across the dataset. These non-unique segments have a unique property called the *degree of repetition* which represents number of times a particular content exist within a dataset.

From the perspective of the data deduplication system, the segments are really divided into segments whose contents must be stored and segments which can be represented only as a reference to an already stored segment. We further classify segments as either the *content segments* or the *duplicate segments* based on whether the content of the segment is required to reconstruct the logical segments. The content segments can be duplicated and reordered to construct the original dataset given the ordering information of contents. The duplicate segments are simply redundant contents whose

TABLE I
DATASETS. SIX DATASETS WITH THE SIZE OF THE LOGICAL SEGMENT SET AND TWO PERFORMANCE METRICS OF INTEREST.

Dataset	Number of Segments	Compression Ratio	Throughput (MB/s)
exchange	1,970,836	0.294	44.72
exchange_is	1,960,555	0.377	42.60
exchange_mb	1,951,271	0.973	27.20
fredp4	1,892,285	0.502	39.78
fredvar	1,922,420	0.209	48.36
workstation	1,966,577	0.403	38.92

only new information is the order of the content.

It is obvious from the two classifications that the duplicate segments must be the non-unique segments while the unique segments are content segments. This is shown in the Fig.2. As shown in the figure, there exists an intersection of non-unique segments and the content segments that are called the *base segments*. The base segments are minimum set of segments whose contents that must be stored to allow the deduplication system to reconstruct all the non-unique segments.

Therefore, the base segments, the unique segments and the duplicate segments are three sets of segments that are disjoint and whose union is the logical segments. A segment at any given time must be a member of one and only one of these sets.

We also define a relational parameter between these sets.

Definition 1 (scr) *Segment compression ratio.* $scr = |C|/|L|$ *where* $L = \{logical\ segments\}$ *and* $C = \{content\ segments\}$.

The scr obviously represents how much of the data is actually redundant in terms of segment count. Typical data deduplication systems have some bounded segment size which ensures that scr closely reflects the actual data reduction.

III. DESCRIPTION OF THE EXPERIMENTAL DATASETS

Six different datasets shown in table I were used in our evaluation. Due to privacy concerns, only the SHA1, size and the backup order of the segments were made available.† We refer to this data as *segment trace*.

The *exchange*, *exchange_is* and *exchange_mb* datasets are all Exchange server data encoded in different ways. The *fredp4* dataset contain a revision control system data and *fredvar* contain data from /var directory in the same machine. The *workstation* dataset contains data from the home directories of several users.

The segment trace is not of the entire original dataset. Unfortunately, the data presented to us were of the data that is truncated from a larger dataset. Although a truncated dataset is used, it does not effect the methodology presented in this paper. This is because we characterize the datasets based on how deduplication system processes them. There are only a specific range of data characteristics from deduplication system's point of view. And these truncated datasets are used to show where in this range a particular dataset may lay. We could easily have done same analysis with the synthetic traces since their characteristics cannot lay outside this range regardless of how we generate the trace.

†The actual data provided also contained compressed size of the segment, stream offset which represents where the segment is located in the original dataset. The compressed size is unused since the local compression of segments is not of interest in this study and stream offset is redundant information which can be deduced from the order and the size of the segments.

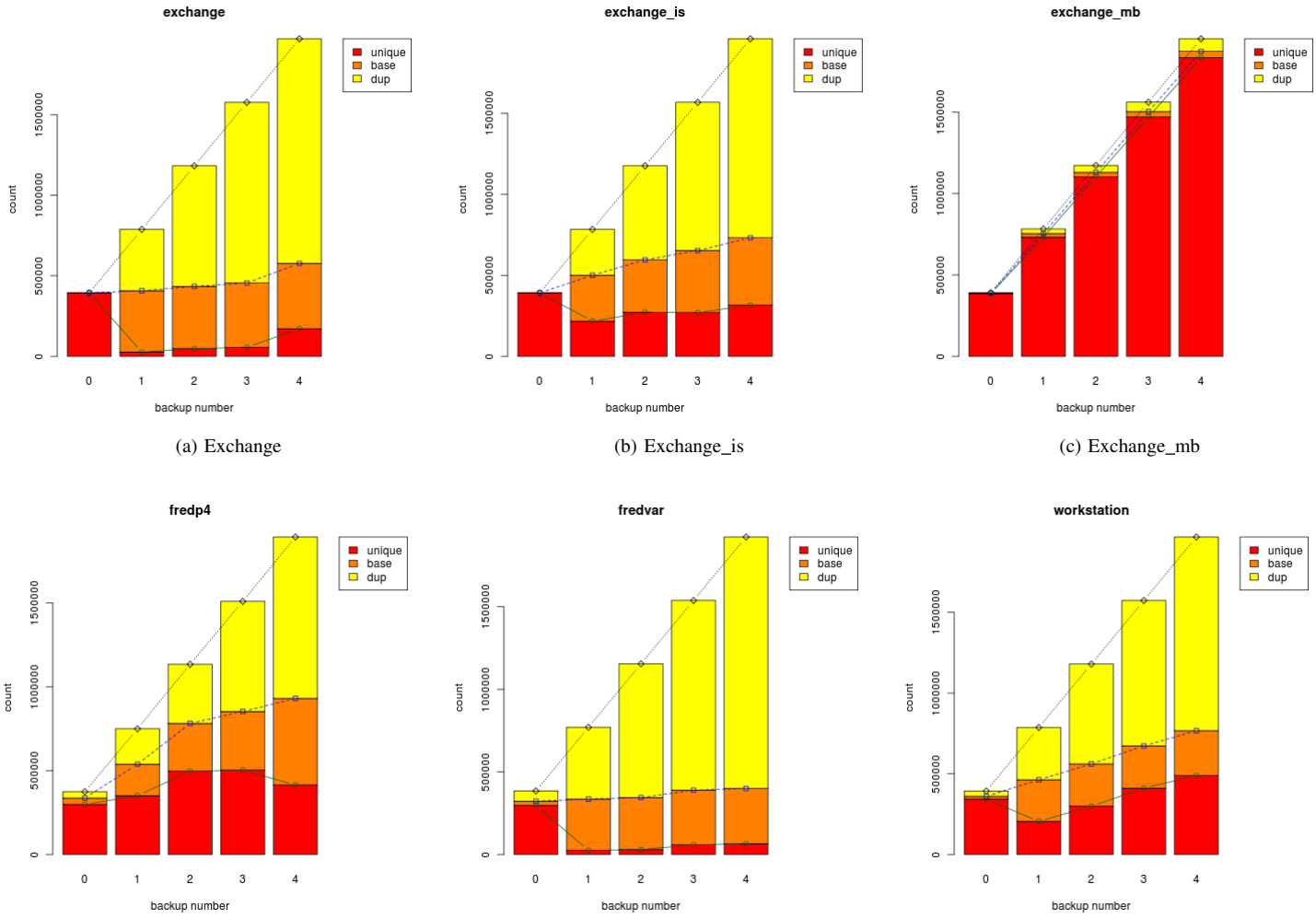
TABLE II
EXPERIMENTAL DEDUPLICATION PARAMETERS.

Parameters	Value	Parameters	Value
Content Generation Method	Content Based	Minimum Segment Size	4KB
Average Segment Size	8KB	Maximum Segment Size	16KB
Storage Maximum Throughput (sequentual)	50MB/s	Maximum Network Throughput	100MB/s
Content Dictionary Cache Size	100,000 entries	Hash Function for Segment ID	SHA1 (160 bits)
# of Bloom Filter Hash Functions	4	Bloom Filter Size	1,000,000 entries

The system parameters for data deduplication system is described in the table II. The cache size for the *content dictionary* is kept reasonably small to capture the effect of cache misses. Since the maximum raw throughput of the system is IO limited at 50MB/s, you can see that the throughput of different datasets vary from being close to the maximum throughput to about 50% of that throughput.

As mentioned earlier, there is a strong linear relationship between the compression ratio and the throughput. The sample correlation coefficient, $r_{ct} = -0.98$ (where $c = compression\ ratio$ and $t = throughput$), suggest that higher compression (lower compression ratio) results in higher throughput. While it may seem that compression ratio is the dominant factor in determining the throughput since the two are so highly correlated, the system for which the throughput numbers are reported are *in-line* deduplication systems where all segments must be compared against the content dictionary before it can be stored on the disk. Therefore, the elimination of duplicate segments not only results in dictionary update but also additional disk writes. In systems where data may already all be written to the disk before being looked up in dictionary may not perform in exactly the same manner. Therefore, we concentrate more on dictionary access pattern to analyze potential throughput concerns.

The segment trace contains 5 weekly full backup information of 6 different datasets and 5GB of each full backup. Therefore, all data presented in table I are of the same size at 25GB. The variation in number of segments is due to variation in average segment size. Compression ratio provided in the table is compression due to data deduplication process only and does not include additional compression provided by local compression of segments using traditional data compression



(a) Exchange (b) Exchange_is (c) Exchange_mb

(d) fredp4 (e) fredvar (f) workstation

Fig. 3. Composition of segments for different datasets. Graphs show how the composition of segments changes over time as weekly full backups are deduplicated.

tools.

IV. Data Characterization

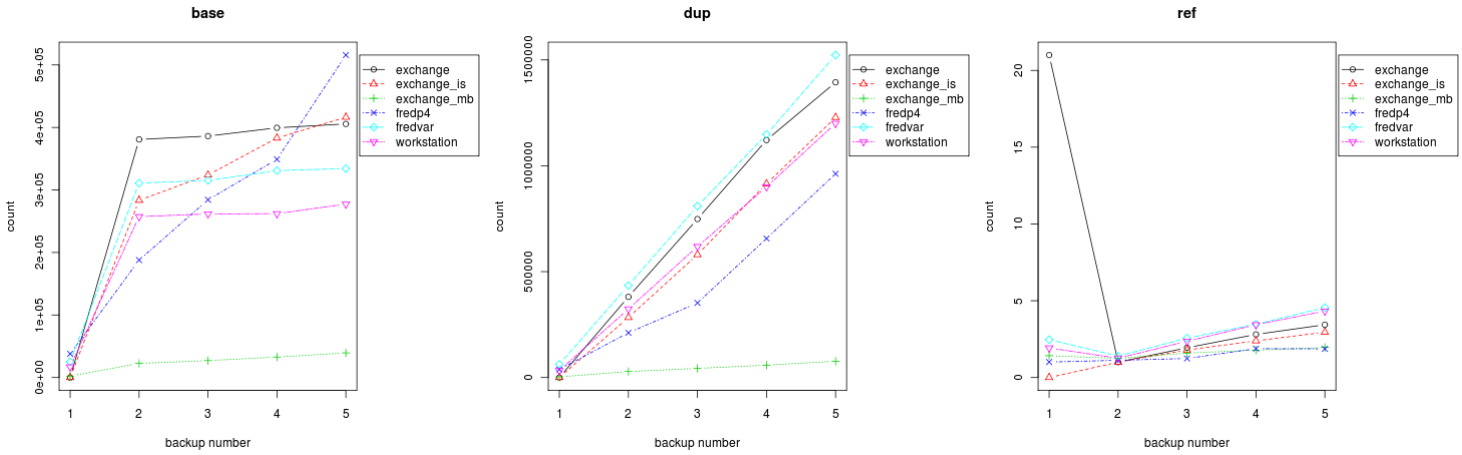
An obvious and perhaps the most important data characteristic for data deduplication is amount of redundancy. Two datasets of equal redundancy would yield in similar compression assuming the content generation was done in a reasonable manner. However, they could perform very differently depending on the other aspects such as locality of segments and fragmentation due to elimination of duplicate segments. The difficulty in characterizing workload of data deduplication is that the characteristics is actually dependent on the contents. While access patterns and physical attributes such as size of files tends to follow a well known distributions in a large scale [20], [21], contents themselves are random in nature due to human factor and different encoding deployed by different applications.

A. Composition of Segments

Fig.3 shows how the composition of segments, as defined in Fig.2, changes over time for each dataset at a full backup granularity. The composition of the segments is quite different for all datasets. In the figure *backup number* represents accumulated number of full backups stored in the system. We make a few observations from the composition.

Observation 1 *Amount of duplication found within a single full backup is negligible compared to the duplication found across the full backups.*

This is an observation also made by other deduplication works on backup data [11]. This observation allows us to look at the changes in the composition of segments as results of deduplicating a full backup instance against other instances of full backup. While non-zero unique and base segments at backup number 0 in Fig.3d, Fig.3e and Fig.3f indicate



(a) Base segments (b) Duplicate segments (c) Average references

Fig. 4. Comparison of data sets. Number of references are simply $|D|/|B|$. Therefore, *exchange_is* and *frep4* show lowest average number of references since their number of base segments increase with the number of duplicate segments. For all other segments, the number of base segments stay relatively steady indicating that same portion of the data is changing over and over again.

that there exists some inter-duplication within a backup, the amount is negligible and we ignore its effect in this paper.

Observation 2 *For the base segments there are two distinctive case where number of base segments increase with number of fulls as in Fig.3b and Fig.3d and case where number of base segments stay steady as shown in Fig.3a, Fig.3e and Fig.3f.*

This observation can be made more pronounced in Fig.4a.

The number of base segments can only increase in the absence of the deletion. Once a segment becomes a base segment, it cannot become any other type of segment. Therefore, the number of base segment can only stay steady if there is no or very little amount of new base segments are created. We can conclude that in this kind of dataset, the unique segments stays unique segments while the same set of base segments keeps getting duplicated.

Conversely, the increase in number of base segments mean that the unique segments of previous backup number is getting converted to the base segments. This happens only when a newly generate data at backup number i still exists at backup number $i+1$. We assume that no base segments are generated within a full backup instance based on the observation 1. We call this rate of conversion *base generation rate*(bgr) which is bounded by the number of unique segments in previous backup number and is larger than or equal to 0.

We formally define bgr below.

Definition 2 (bgr) *Rate at which unique segments are converted to base segments.* $bgr_i = \frac{|B_{i+1}|-|B_i|}{|U_i|}$ *where the subscripts represent the backup number. We define* bgr *to be average value of* bgr_i, $bgr = \frac{\sum bgr_i}{max(backupnumber)}$.

Observation 3 *The number of unique segments either increase steadily as shown in Fig.3c and Fig.3f or stay relatively steady as in the rest of the figures in Fig.3. While the number of unique segments can decrease, it does not happen very often.*

The number of unique segments, $|U|$, at backup number i is increase by amount of unique segments generated by that particular full backup and is decreased by amount of unique segments at backup number $i-1$ converted to base segments[‡] at backup number i. Since the amount of decrease in $|U_{i+1}|$ is $|U_i| \times bgr_i$, we also need to define a parameter to represent the increase in $|U|$, *unique segment ratio*(usr). This ratio does not represent the changes in the number of unique segments, $|U|$, since there is also conversion of segments from unique to base.

Definition 3 (usr) *Relative amount of unique segments within a full backup.* $usr_i = \frac{|U_{i+1}|-|U_i|(1-bgr_i)}{|L_{i+1}|}$. *We define* usr *to be average value of* usr_i, $usr = \frac{\sum usr_i}{max(backupnumber)}$.

From the definition of usr it is obvious that we cannot simply determine the rate of unique segment generation from looking at the changes in $|U|$. However, since it is possible to determine the bgr from also looking at the changes in number of base segments, $|B|$ which in turn allow us to calculate usr. Since increase in $|B|$ is only possible due to bgr, if increase is very small, we can assume that $bgr \simeq 0$. This allows us to think that $usr_i = U_{i+1} - U_i$ which means that unique segments in backup number i no longer exists in backup number $i+1$ or else they would have become base segments. This pattern is most pronounced in Fig.3f. Intuitively, the pattern suggests the existence of a small and heavily updated *working set* within the file system as suggested by [22]. Similar analysis could applied to Fig.3c where almost the entire dataset is a working set.

[‡]Conversion to duplicate segments is also possible but is ignored due to observation 1.

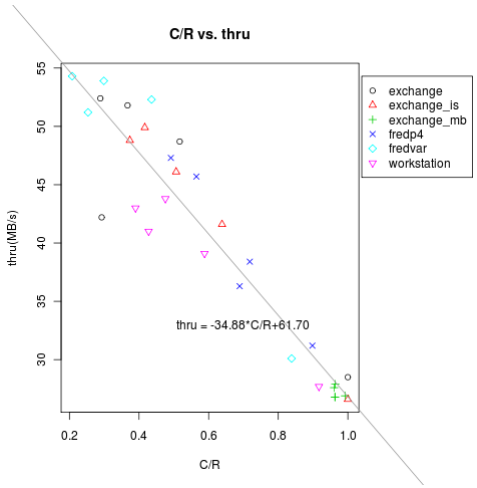


Fig. 5. C/R vs. Throughput. The accumulated compression ratio of each dataset at each full backup is plotted against the throughput for that particular backup window. You can observe that the throughput increases as the accumulated compression ratio decreases even though every full backup is equal in size.

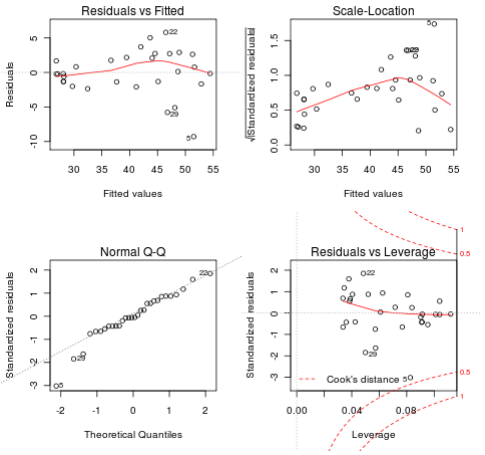


Fig. 6. Analysis of C/R vs.throughput. The top left graph shows that while the regression is a good fit, data around throughput value of 40MB/s to 50MB/s tends to out perform expected throughput. Top right graph shows square root of residual error. The backup instance 5, 22 and 29 show a large deviation (from exchange, fredvar and workstation datasets respectively). The lower left graph shows that the residual errors are more or less normally distributed with the exception of 4, 21, 29 (from exchange, exchange_is and workstation datasets respectively). The lower right graph shows the Cook's distance. Backup instance 5 is less than 0.5 Cook's distance away and can be safely taken into evaluation.

Cases where $|B|$ actually increase with the backup number, bgr represents the rate at which new data becomes part of *non-working set*. It suggest that part of the working set becomes stable over time and becomes base on which future segments are deduplicated. Extreme case is shown in Fig.3d where the effect of bgr starts to outweigh the effect of usr and the number of unique segments actually decrease. This is the most desirable case for the data deduplication where amount of data to be store grows sub-linearly and therefore more scalable in terms of backups you can store. It will also be shown that the throughput also tends to be higher in these cases for similar compression.

Cases seen in Fig.3a and Fig.3e suggest that there exists only a small change between the backups. While this is also a desirable case for data deduplication, it may also be argued that for a storage system with such little activity, a longer period of incremental backups could provide similar performance.

The logical size, usr and bgr of a dataset completely describes the dataset as long as there is no deletion involved. Furthermore, they do not vary as much as dcr over time. Therefore a aggregated values of usr and bgr are much better choices of parameters to describe a dataset. They show how much of new data is generated ($|L| * \overline{usr}$), how much of that data will be changed again before the next full ($1 - bgr$). Together, they allow you to calculate the compression ratio at time $i + 1$ by following these steps.

1) $|B_{i+1}| = |U_i| * bgr_i + |B_i|$.
2) $|U_{i+1}| = |L_{i+1}| * usr + |U_i|(1 - bgr_i)$.
3) Calculate $|D_{i+1}|$ and scr from above two values.

B. Segment Run Length

In section III, we showed that the compression ratio and the throughput of a deduplication system can be highly correlated. Fig.5 shows throughput vs. compression ratio of each backup instances. This graph together with Fig.6 shows that the compression ratio is indeed a most relevant factor in determining throughput. However, Fig.6 shows some skewness both in Residual vs. Fitted graph and Residual vs. Leverage graph. They suggest at other factors that affect throughput which are not simply white noise.

Fig.5 also shows a linear estimation line. One interesting observation is that there are some datasets which perform consistently better than the estimation. These are *exchange_is*, *fredp4* and *fredvar* datasets. On the other hand, *workstation* dataset performs consistently worse than expected.

To quantify this effect, we take look at the deviation of throughput numbers from the regression line as shown in table III. These numbers are not absolute deviation. These are average values of $actual\ throughput - expected\ throughput$ and removes the effect of the compression ratio from the throughput numbers. As explained in section II, the *content dictionary* is the only place where this variation in performance due to content could occur. Note that all data are of the same size, written sequentially once. While *content store* actually stores less if more segments are deduplicated, this performance difference is likely to be linear to the amount of data stored. Since we have removed this linear factor and concentrate only on the deviation, the effect must be due to on content dictionary behavior.

To evaluate the content dictionary behavior, we first look at the sequential runs of segments. A longer sequence of a run represents a sequential access of content dictionary making it easier for the dictionary cache to be a hit. A *run* is simply a sequence of consecutive segments that are either unique or

TABLE III
AVERAGE DEVIATION FROM THE EXPECTED THROUGHPUT AND THEIR CORRESPONDING RUN LENGTH INFORMATION

Dataset	Average Deviation	Max Run Len.	Average Run Len.	Run Len. > 1000
fredp4	1.532	207400	7.944	57%
exchange_is	1.375	392000	15.88	27%
fredvar	0.852	366200	28.88	76%
exchange	0.217	183100	30.53	64%
exchange_mb	-0.722	7490	18.6	13%
workstation	-3.260	5632	6.635	9%

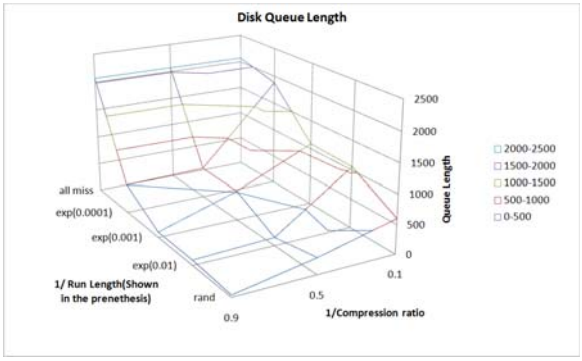


Fig. 7. Run length vs. compression ratio vs. queue length.

duplicate segments. We refer to them as a *content run* and *duplicate run* respectively when need to distinguish between them. Table III shows aggregated *run* data. We make following observations.

Observation 4 *The average run length varies little across the datasets and shows little correlation to the throughput. However, the maximum run length varies significantly across the datasets and show high correlation.*

The correlation coefficient of maximum run length and the average deviation is $r^2 = 0.63$. It is high enough that we can safely assume there is significant correlation between the two values. The correlation coefficient of average run length and the average deviation is only $r^2 = 0.002$. The obvious reason is that the mean values of run length does not represent anything physical when the significant portions of the data lay in outlier regions. The last column of table III shows that the over 50% of the segments fall in a run length of over 1000. It is interesting to note that the *fredvar* and *exchange* datasets suffer in throughput even though they have the highest percentage of large runs. It leads us to our next observation.

Observation 5 *Datasets with high average run length due to many extremely large runs perform worse than cases where the run length are more evenly distributed.*

The skewed run length has a negative impact on performance not because of the content dictionary but the content store. While it is obvious that less amount to be stored result in high logical throughput as described earlier, the arrival rate of segments to the content store is also important. In a simple queuing model of Fig.1. Since the arrival rate at the content store is simply $(1-p)S$ where S is the throughput of the content dictionary and p is content dictionary hit probability. If p was uniformly distributed, that the arrival rate would simply be a function of compression ratio. However, as the skewed distribution of run length shows, p of each segments are highly correlated. The effect is bursty arrivals at the content sore resulting in high queue length and lower throughput as shown in Fig.7. It is shown that the effect of run length is less pronounced for datasets with higher compression ratio. The *all_miss* case is the worst case where every segment is sent to the segment store and *rand* is the best case where run length distribution is uniform.

The run length analysis shows that while the throughput is highly correlated with the compression ratio, higher run length typically results in higher throughput. Additionally, skewed run length which typically results from few very long runs, results in lower throughput especially when the compression ratio is low.

V. APPLICATIONS

Above analysis of composition of segments allows users of deduplication systems to collect specific statistics to predict future storage requirements. In the world of deduplication where the physical storage space and the logical segment space does not match, it is difficult to provision storage space. Extracting bgr and usr from the deduplicated data allow users to better predict future storage requirements.

For the given six test datasets, we use bgr and usr of the first three fulls to predict the comression ratio of 4th and 5th full and compare them with the dcr approach.

For example, the $bgr_1 \sim 0.01$ and $usr_1 \sim 0.26$ for *workstation* dataset. Since $|L_i| \sim 393311$, we can predict future segment composition given backup number 2 information. Fig.8 shows the graph of the original workstation composition and the graph of predicted composition. Since the parameters were extracted from the backup number 1 and 2, the accuracy of prediction falls as the number of fulls increase. Of course, the accuracy of the prediction itself is also dependent on the dataset and the worst case error was observed in *exchange* dataset where the scr at backup 4 was off by just over $0.02(\sim 5\%$ error). This is much better approach than simply observing the compression ratio. For *fredp4* dataset the dcr prediction is off in compression ratio by 0.18 ($\sim 36\%$ error).

Table IV shows our approach outperforms the tradistional dcr approach in all but exchange_mb dataset where the error is about the same.

Another potential application is evaluating the impact of merging two separately deduplicated data. Unless there is a reason to suspect a large similarity between the two datasets, parameters of each dataset can be superpositioned to predict the composition of combined dataset.

TABLE IV
COMPRESSION RATIO PREDICTION ERROR COMPARISON

datasets	dcr	usr & bgr
fredp4	35.99%	3.68%
exchange_is	9.20%	2.21%
fredvar	7.35%	0.21%
exchange	17.82%	5.42%
exchange_mb	0.14%	0.20%
workstation	3.61%	0.61%

A more involved application would be to filter out the *working set* from being deduplicated. Various techniques exists to evaluate the working set of a storage system [23], [24] which can be used to identify *hot data* and pass only those deemed cold to the deduplication system. This would be especially useful for datasets with low bgr where same portions of the data are constantly changing. This portion can deemed unworthy of deduplication and is backed up separately.

The analysis of run length suggest that it maybe more beneficial for the through put if the inter-duplicate segments are ignored. That is we only evaluate duplicate segments generated between the full backups. These single or dual duplicate segments fragment the index table and potentially the contents on disk without providing any substantial gain in compression.

Last application maybe to adjust buffer sizes at each stage of deduplication based on the run length observed to minimized skewed arrival effect.

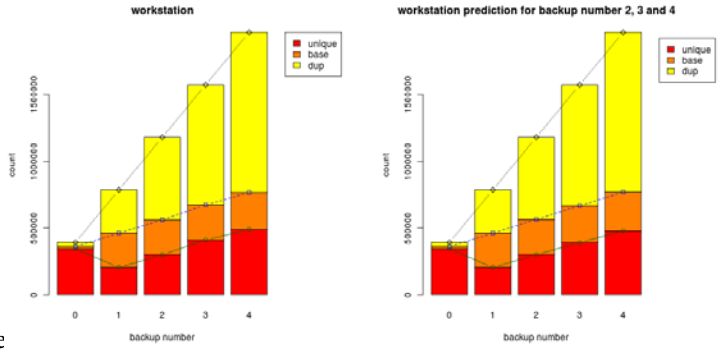
VI. Related Work

The workload presented in typical deduplication systems paper are either from a privately accessible systems [1], [2], [6], [10], [16], [17], [25] or a relatively small dataset of specific type in public domain [11], [12], [26]. However, either techniques allow a direct comparison of different systems if the dataset under evaluation is completely different or if it only covers a dataset of a particular characteristics.

There have been few work in trying to statistically characterize the input dataset to various benchmark programs [27]–[29]. However, these papers concentrate on generic sub-setting of dataset so that different characteristics of application is explored statistically hence lacking in accuracy when it comes to predicting exact performance variances.

Numerous works have also been applied in filesystem activities and its contents [21], [30], [31] with one paper using a simple statistical analysis to characterize a filesystem impression [32]. While these approaches all reveal some interesting intuitions of how we use storage systems, the scope is too generic to be neither accurate or simple enough to be applied in real systems. More specific workload analysis have been targeted to SSDs [22] and we use some of their approach in out analysis.

Moreover, we believe that this is a first approach in analyzing the deduplication process in terms of the input dataset.



(a) Real Data (b) Predicted Data

Fig. 8. The original segment composition of workstation and the predicted values.

VII. Limitations

Many deduplication systems allow the segments to be stored without duplicate detection for the better throughput [6], [14]. This does not cause correctness issue. However, it makes predicting performance even more difficult. In this study we assume that all duplicate segments must be identified and eliminated. There has also been attempts to make better decisions on identifying the data segments from the dataset [3], [11], [12]. We claim that the segments, regardless of how they are defined, define datasets in terms of data deduplication process. This claim assumes that different datasets will have statistically similar relative compression and performance regardless of the segmentation process. In other words, segmentation process affects different dataset in statistically similar way. For example, if dataset A gained compression ratio by 10% while losing throughput by 5% by using the new segmentation method, than dataset B is highly probable to gain in compression and lose in throughput by similar amount. While this assumption is actually verifiable, lack of datasets prevents us from providing strong statistical guarantee. However, previous analysis of segmentation algorithms [11] as well as our own tests indicate no evidence to reject our assumption.

VIII. Conclusion

We have shown a framework for characterizing datasets by analyzing how different datasets behave within a data deduplication system. For the datasets presented here, we show that there are classes of datasets which behave similarly.

We also show that the changes in the number of base segments is more important in terms of scalable data compression than the simple compression ratio. More specifically, a positive bgr guarantees continuing reduction in the compression ratio resulting in sub-linear growth in the storage requirement. Intuitively, bgr represents life time of new data. High bgr means that newly created data are long lived and low bgr means that they are short lived. Since usr represents amount

of new data created, their ratio tells you how much of the new data is transient and how much are more permanent. Generating more and more permanent data means more data to be deduplicated at each back up resulting in better compression ratio.

We also show that the segment run length has a fundamental effect on the dictionary lookup process both in terms of the cache behavior and queuing delays, which can counteract each other in terms of throughput. More specifically, the typical belief that high average run length results in better throughput is not true. In fact, it can lower the throughput due to skewed arrival rate the content store. However, high maximum run length tends to improve performance due to good caching effect of content dictionary. Simply put, one can estimate throughput based on the compression ratio using a C/R vs. throughput regression line for a given system and can further expect the throughput to be better or worse based on the run length information.

Lastly, we showed that we can increase the accuracy of compression ratio prediction by as much as 80%. Furthermore, we can accurately predict the composition of segments which allow us to determine the existence of frequently updated data which does not have to be deduplicated.

ACKNOWLEDGMENT

The authors would like to thank Fred Douglis and Grant Wallace at EMC for their invaluable comments and allowing us to run experiments on their datasets.

REFERENCES

[1] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. New York, NY, USA: ACM, 2009.

[2] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2008.

[3] D. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 4, p. 448, 2006.

[4] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 174–187, 2001.

[5] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. New York, NY, USA: ACM, 2009.

[6] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: large scale, inline deduplication using sampling and locality," in *Proceedings of the USENIX conference on File and storage technologies*. Berkeley, CA, USA: USENIX Association, 2009.

[7] L. You, K. Pollack, and D. Long, "Deep Store: An archival storage system architecture," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, 2005, pp. 804–815.

[8] M. Smith, J. Pieper, D. Gruhl, and L. Real, "IZO: applications of large-window compression to virtual machine management," in *Proceedings of the 22nd conference on Large installation system administration conference*. USENIX Association, 2008, pp. 121–132.

[9] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. New York, NY, USA: ACM, 2009.

[10] A. Clements, I. Ahmad, M. Vilayannur, J. Li, I. VMware, and M. CSAIL, "Decentralized deduplication in san cluster file systems," in *Proceedings of the USENIX Annual Technical Conference*, 2009.

[11] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal Content Defined Chunking for Backup Streams," in *Proceedings of the Eighth USENIX Conference on File and Storage Technologies*, 2010.

[12] K. Eshghi and H. Tang, "A framework for analyzing and improving content-based chunking algorithms," *Hewlett-Packard Labs Technical Report TR*, vol. 30, 2005.

[13] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication," in *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, ser. Companion '08. New York, NY, USA: ACM, 2008.

[14] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009.

[15] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, 1970.

[16] R. Koller and R. Rangaswami, "I/o deduplication: Utilizing content similarity to improve i/o performance," in *Proceedings of the 8th conference on file and storage technologies (FAST '10)*, 2010.

[17] S. Rhea, R. Cox, and A. Pesterev, "Fast, inexpensive content-addressed storage in foundation," in *Proceedings of the USENIX Annual Technical Conference*, 2008.

[18] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The Zettabyte File System," in *FAST 2003: 2nd Usenix Conference on File and Storage Technologies*, 2003.

[19] M. Dillon, "The Hammer Filesystem," 2008.

[20] A. Riska and E. Riedel, "Evaluation of disk-level workloads at different time scales," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, pp. 67–68, 2009.

[21] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008.

[22] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD Lifetimes with Disk-Based Write Caches," in *Proceedings of the USENIX Conference on File and Storage Technologies*. FAST, 2010.

[23] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim, "FlexFS: A Flexible Flash File System for MLC NAND Flash Memory," in *Proceedings of the USENIX Annual Technical Conference, San Diego, CA*, 2009.

[24] W. Wang, Y. Zhao, and R. Bunt, "HyLog: A High Performance Approach to Managing Disk Layout," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. USENIX Association, 2004, pp. 144–158.

[25] U. Manber *et al.*, "Finding similar files in a large file system," in *Proceedings of the USENIX winter technical conference*, 1994.

[26] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, "Secure data deduplication," in *Proceedings of the 4th ACM international workshop on Storage security and survivability*, ser. StorageSS '08. New York, NY, USA: ACM, 2008.

[27] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere, "Quantifying the impact of input data sets on program behavior and its applications," *Journal of Instruction-Level Parallelism*, vol. 5, pp. 1–33, 2003.

[28] J. J. Yi, D. J. Lilja, and D. M. Hawkins, "A statistically rigorous approach for improving simulation methodology," in *in Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, 2002.

[29] W. C. Hsu, H. Chen, P. C. Yew, and D.-Y. Chen, "On the predictability of program behavior using different input data sets," in *Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, ser. INTERACT '02. Washington, DC, USA: IEEE Computer Society, 2002.

[30] J. Douceur and W. Bolosky, "A large-scale study of file-system contents," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1, p. 70, 1999.

[31] J. R. Douceur and W. J. Bolosky, "A large-scale study of file-system contents," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, pp. 59–70, 1999.

[32] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Generating realistic impressions for file-system benchmarking," *Trans. Storage*, vol. 5, pp. 16:1–16:30, December 2009.

Performance Characterization and Acceleration of Optical Character Recognition on Handheld Platforms

Sadagopan Srinivasan, Li Zhao, Lin Sun, Zhen Fang, Peng Li, Tao Wang,
Ravishankar Iyer, Ramesh Illikkal, Dong Liu
Intel Corporation

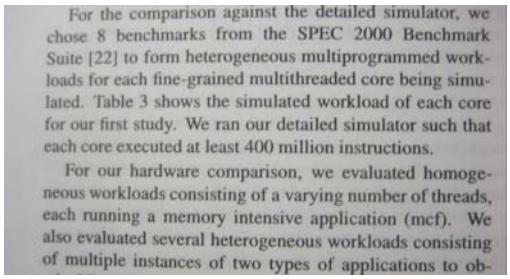
Abstract *— Optical Character Recognition (OCR) converts images of handwritten or printed text captured by camera or scanner into editable text. OCR has seen limited adoption in mobile platforms due to the performance constraints of these systems. Intel® Atom™ processors have enabled general purpose applications to be executed on handheld devices. In this paper, we analyze a reference implementation of the OCR workload on a low power general purpose processor and identify the primary hotspot functions that incur a large fraction of the overall response time. We also present a detailed architectural characterization of the hotspot functions in terms of CPI, MPI, etc. We then implement and analyze several software/algorithmic optimizations such as i) Multi-threading, ii) image sampling for a hotspot function and iii) miscellaneous code optimization. Our results show that up to 2X performance improvement in execution time of the application and almost 9X improvement for a hotspot can be achieved by using various software optimizations. We designed and implemented a hardware accelerator for one of the hotspots to further reduce the execution time and power. Overall, we believe our analysis provides a detailed understanding of the processing overheads for OCR running on a new class of low power compute platforms.*

1. Introduction

Smart phones and handheld devices have gained widespread popularity by placing compute power and novel applications conveniently in the hands of end users. iPhone, iPad and Blackberry serve as good examples for this usage trend [18][19]. The introduction of new low-power general-purpose processors like Intel's Atom™ processor family enables future handhelds to enjoy a larger base of general-purpose applications.

One of the emerging consumer electronics applications that have entered the mobile domain is Optical Character Recognition (OCR). OCR converts images of handwritten or printed text captured by camera or scanner into electronic text. It has been widely used in database systems, where scanned books and other document materials are converted into text so that they can be accessed conveniently by search engines. While OCR continues to be used for database and search engines in desktop/server platforms, it is gaining traction in mobile platform as well [3]. One such usage model for OCR in mobile domain is a reading device, which can help people with vision and reading problems to read a book or text by reading it out to them. Few such examples are KNFB a Reader Mobile which is loaded onto Nokia cell phones and Intel Reader [1][17].

Following is an instance of OCR usage model. A person is reading a book or magazine using a smart phone or handheld device. The person can take a picture of the reading material as shown in Figure 1(a). The handheld device recognizes the text in the image and converts it into text as shown in Figure 1(b). Once the text is available, it can be read out to the reader or even be translated to other languages as in [2]. Though image capture and text post-processing are also involved in this usage model, we focus on OCR due to its performance constraints.


For the comparison against the detailed simulator, we chose 8 benchmarks from the SPEC 2000 Benchmark Suite [22] to form heterogeneous multiprogrammed workloads for each fine-grained multithreaded core being simulated. Table 3 shows the simulated workload of each core for our first study. We ran our detailed simulator such that each core executed at least 400 million instructions.

For our hardware comparison, we evaluated homogeneous workloads consisting of a varying number of threads, each running a memory intensive application (mcf). We also evaluated several heterogeneous workloads consisting of multiple instances of two types of applications to ob-

(a) An image of the reading material

for the comparison against the detailed simulator, we chose X benchmarks from the SPEC 2000 Benchmark Suite [22] to form heterogeneous multiprogrammed work loads lor each line- grained multithreaded core being simu lated. Table 3 shows the simulated workload of each core for our first study. We ran our detailed simulator such that each core executed at least 400 million instructions.
For our hardware comparison, we evaluated homoge neous workloads consisting of a varying number of threads, each running a memory intensive application (mci). We also evaluated several heterogeneous workloads consisting of multiple instances of two types of applications to ob

(b) Text file of the above image

Figure 1. OCR Example Illustration

Although OCR has been researched extensively, most of the work has focused on text recognition algorithms [4][5][6][7]. Few works, such as [23], focus on acceleration of Arabic script for DSP processors. As OCR becomes a popular workload for handhelds, where both power and performance are major concerns, it is important to understand its performance characteristics so that architectural improvements can be made to improve user experience. We address this by analyzing the compute and memory requirements of an OCR workload on an Intel® Atom platform.

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

In this paper, we take a reference software implementation [8] of an OCR application and characterize its various phases. We take the above OCR software, analyze the compute requirements, identify the key hotspot functions and propose software optimizations. To the best of our knowledge, this is the first work with detailed performance analysis of OCR workloads on handheld platforms. The main contributions of this paper are:

1. Detailed compute, cache and memory characterization of an OCR software.
2. Analysis of the key hotspot functions of OCR in terms of their scaling behavior.
3. Analysis of performance optimizations such as compute, multi-threading, and sampling step optimizations.
4. Design a hardware accelerator for a hotspot in OCR.

The rest of this paper is organized as follows. We give an overview of an OCR system (OCRopus)[8] and describe the algorithm phases used in Section 2. Section 3 describes the profiling characteristics of OCRopus software on Atom platform and analyzes the key hotspot functions. Section 4 describes a set of software optimizations that we implemented and show their associated performance benefits. Section 5 describes the hardware accelerator for a hotspot. We conclude in section 6 by outlining the direction for future work on this topic.

2. OCR Application Overview

Several OCR implementations, including commercial versions, like Abby [9], ExperVision TypeReader & OpenRTK [10] have been widely used by the text recognition community. We chose OCRopus 0.3 for our study due to the following factors: 1) OCRopus is open source software, 2) it supports modularity and reuse across various phases and, 3) we believe that although commercial solutions may have a different recognition algorithm, the basic phases and main functionalities of OCR system stays roughly the same as OCRopus.

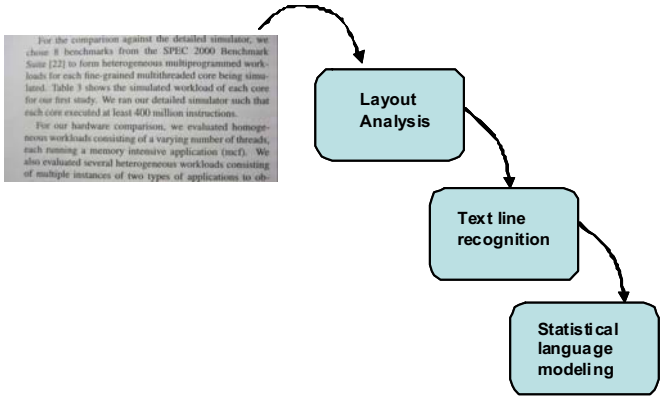


Figure 2. OCRopus execution flow

OCRopus has four major phases:

(a) Pre-processing
(b) Layout analysis
(c) Text line recognition
(d) Statistical language modeling

Pre-processing: This phase includes binarization (converting a grey-scale or colored image into binary image), image smoothing and skew correction. This phase has a direct impact on OCR.

Layout analysis: This phase uses the binarized image generated earlier and segments the image into non-text regions, text columns, text blocks and text-line regions. For each region, features like proximity or white space are extracted and used to classify regions into text and white space region. OCRopus uses RAST-based layout analysis, which includes column finding, text line finding and reading order determination [11].

Column finding identifies whitespaces using a maximal whitespace rectangle algorithm to find vertical whitespace rectangles with a high aspect ratio. The rectangles that are adjacent to character-sized components on its left and right have high likelihood to be column boundaries. Figure 3 shows the input and output of layout analysis phase. The text in the input image is bounded by boxes, the paragraphs are segmented from each other and, columns are identified and demarcated.

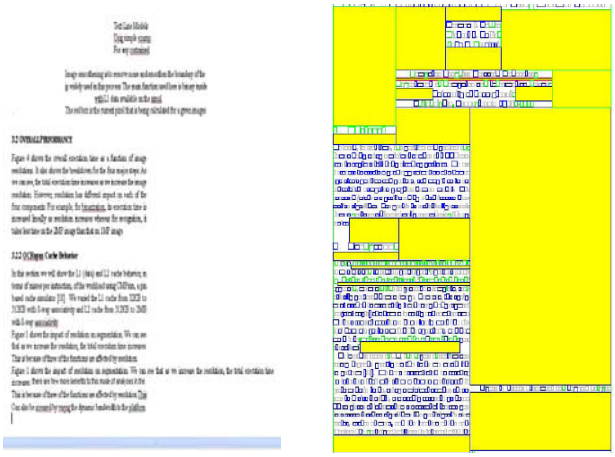


Figure 3 (a). Input image **(b). Layout output**

Text line recognition: In this phase, text lines from the previous phase are used to recognize the text within each line. OCRopus incorporates *Tesseract* as the text line recognition engine [6].

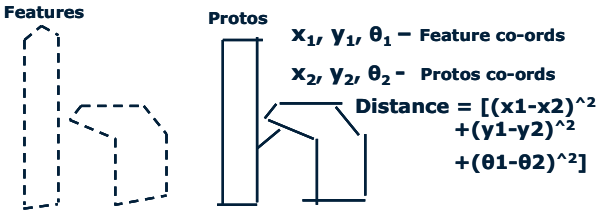


Figure 4. Tesseract recognition phase

Tesseract uses a two-pass shape comparison between *protos* character shapes in the database and *features* identified in the input image. In the first pass, the words that are recognized with good accuracy are passed to an adaptive classifier as training data. As the classifier is being updated to adapt to the

input words, it will do a better job for text lines that are down the page. Figure 4 shows the computation involved in the recognition phase between the features identified and the protos for character 'h'. The co-ordinates x, y represents the row and column position of the *features*/*protos,* and **θ**(theta) is the angle.

Statistical language modeling: This phase resolves ambiguous characters obtained in the previous phase. Statistical language models can be dictionaries, stochastic grammars, etc. OCRopus uses open source OpenFST library [12] as its language modeling tool. This phase improves the accuracy of text recognition.

2.1 Components that we study

Among the four major phases, the language modeling phase is optional in OCRopus. Therefore, we only focus on the first three phases in this paper. While pre-processing is supposed to be the very first step, some of its functions are scattered among other phases for performance efficiency. For example, the image smoothing for noise reduction is performed after layout analysis as this function is only needed for text line regions instead of white space. Therefore we breakdown the OCRopus code into four components: *binarization*, *segmentation* which represents the layout analysis phase, *image smoothing* and *text recognition*. We will focus on these four components and their performance in this paper.

3. OCR Application Performance on Atom-based Platform

In this section, we analyze OCRopus implementation running on the Atom platform. We start with an overall performance analysis and then dive into each components.

3.1 Platform Configuration and inputs

Our measurement is based on a 1.6 GHz Atom processor running CentOS 4.1 Linux kernel 2.6. The CPU had hyper-threading enabled supporting 2 hardware threads. The core has 24KB L1 data cache with 6-way associativity, 32KB instruction cache with 8-way associativity, and 512KB unified L2 cache with 8-way associativity. The front-side bus runs at 400 MHz and is connected to 1GB of DDR2-533 DRAM.

The performance of OCRopus is dependent on various image parameters as listed in Table 1. Although other factors such as skew degree and clarity can affect performance, these four factors are the important and representative features of images that allows us to understand the OCRopus behavior. We used the sample input images from the OCRopus package as well as images taken by us of random text pages. Only images with very high recognition accuracy are considered for this study.

Table 1. Input Image Parameters

Factor	Values
Image Resolution	1 - 15MegaPixels(MP)
Number of text lines	1, 5, 10, 100, 200
character pixels/total pixels ratio	30-60%

3.2 Overall Performance

Figure 5 shows the overall execution time as a function of image resolution. It also shows the breakdown for the four major steps. The base input image has 5MegaPixels(MP) resolution and we scaled down the same image to various resolutions for this experiment.

This study shows that though the total execution time increases with the image resolution, it has different impact on each of the four components. *Binarization* and *smoothing* time increases linearly with resolution, whereas recognition phase stays almost the same for all but 1MP image. As mentioned earlier, there are various factors besides resolution that impact the execution time. Therefore in the following subsections, we analyze each component in detail and show its scaling behavior.

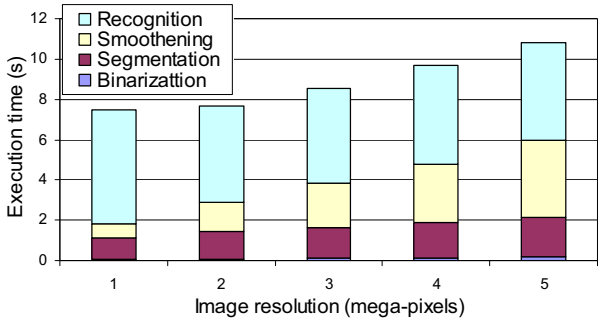


Figure 5. Execution time breakdown

Figure 5 also shows that among the four components, binarization takes less than 2% of the total execution time. Binarization phase converts the grey-scale or colored image into a binary image. It computes a threshold based on the average pixel value of the entire image and sets each pixel in the binarized image to 0 or 1 by comparing its original value to the threshold. Hence, the higher the image resolution, the more time is spent on Binarization. However, this phase constitutes a negligible part of the total execution time and is small compared to the other three components. Therefore we will focus on the other phases for scaling behavior studies.

3.2.1 Scaling Behavior

In this section, we measure the execution time of each OCR component and analyze its scaling behavior as a function of the parameter it depends on. Since the various OCR components scale differently with respect to resolution and text content, we characterize each phase independent of the remaining phases.

1) Segmentation

Segmentation, which represents the layout analysis phase, consists of four main functions as listed in Table 2. This is a compute intensive phase as seen from Figure 5. Each input image factor has a different impact on these functions; therefore segmentation is affected by multiple factors in varying degrees. We characterize these factors separately based on the functions.

Table 2. Major functions in Segmentation

Function name	Description
Label component()	Identifies text pixels that are connected to each other.
Bounding box()	Finds the maximal whitespace rectangle that separates fringe area from content
Compute whitespace()	Identifies the columns and other whitespaces in the text
Extract()	Extracts text lines from columns and assigns the reading order

Figure 6 shows the impact of resolution on segmentation. The execution time of segmentation increases with resolution for the two functions: label_component and bounding_box. Both these functions operate on every individual pixel for their respective process. The other two functions remain almost constant with resolution.

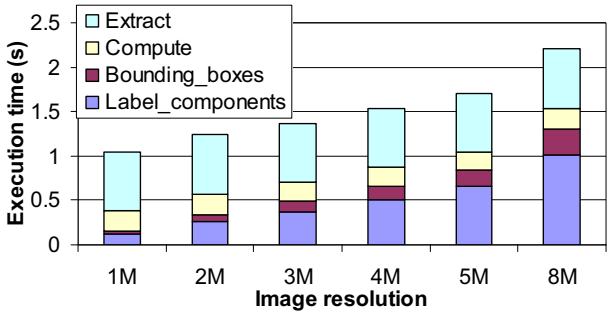


Figure 6. Impact of image resolution on Segmentation

To understand the scaling behavior of the remaining functions in segmentation, we used a metric called character to total pixels ratio. This represents the text content in an image. *Character* pixels are the pixels that are classified to be part of a character (data) during the segmentation phase and the rest are classified as *non-data* pixels. These non-data pixels are separate from the image area that is classified as column boundaries. We used this metric as we observed that different font sizes can lead to different amount of text content in an image and, OCR processing was dependent on pixels rather than actual content.

We fixed the image resolution at 8 MP for this study and varied the ratio of character to total pixels by increasing the data content in the image. The results are shown in Figure 7. As expected, the first two functions were not affected by the image content as the resolution was fixed. As we increased the character-total pixel ratio, the execution time for *compute()* function decreased while that of *extract()* increased. As described in Table 2, *compute()* function examines non-data pixels alone and searches for the maximal whitespace rectangle to identify columns. Therefore the increased whitespace leads to increased execution time. On the other hand, *extract()* function is dependent on text regions i.e. the actual character content in the image. Hence its execution time increases with the text content (i.e. character pixels).

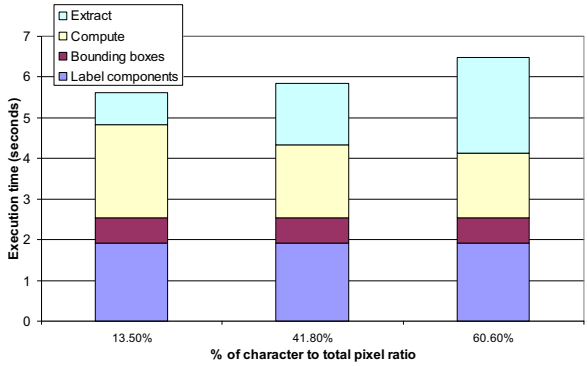


Figure 7. Impact of character-total pixels ratio on Segmentation

2) Image smoothing

Image smoothing removes noise and smoothens the boundary of the characters. Binary morphology [13][14] is widely used in this process and, the main function used in this implementation is *binary dilation*. This function calculates the maximal value for each pixel for a given radius. As shown in Figure 8, each box is representative of a pixel. The central red box is the current pixel that is being smoothed. When the radius is set as three, all the yellow pixels fall within a circle with the red box as its center. The maximum of these 29 values is calculated and used as the value for the new image.

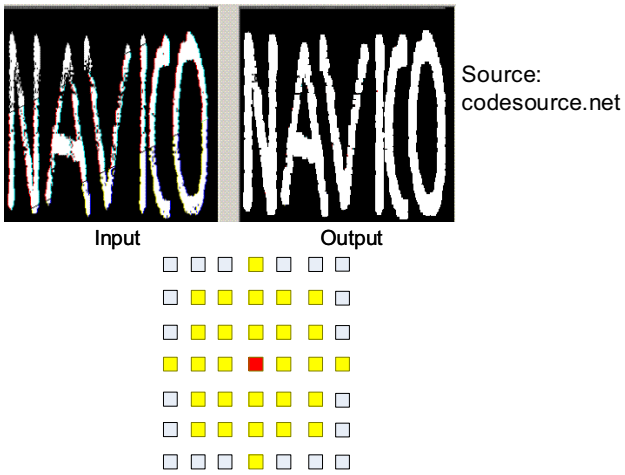


Figure 8. Image smoothing process

Figure 9 shows the impact of image resolution on image smoothing process. We scaled down a 15MP image down to 1MP for this study. We can observe that the execution time scales linearly with the resolution. Although we show the resolution from 1 to 15 mega-pixels for our scaling study, we have observed that this trend is repeated at intermediate resolutions as well as *smoothing* is applied for each pixel of the segmented image. Therefore this step takes a significant amount of time in the total processing time.

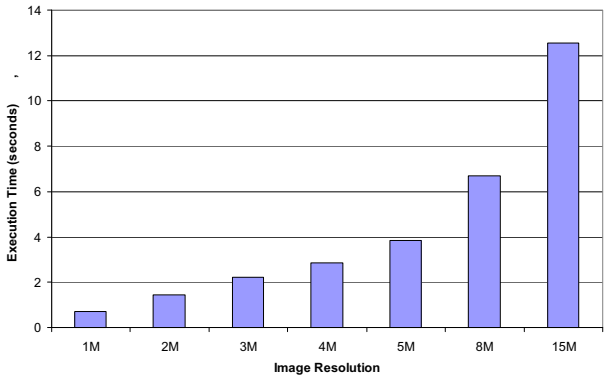


Figure 9. Execution Time Scaling of Image Smoothing

3) *Recognition*

Recognition time in OCR depends on various parameters such as text content reflected by the character-to-total pixel ratio parameter in our experiments, text lines in the input image and clarity of the image, etc. In our studies, we do not consider clarity as there is no good metric to define it. Figure 10 shows the execution time for sample images for various text lines, varied from 1 to 5. We observe that, with other parameters such as image resolution being constant, the recognition time is directly proportional to the number of lines and scales linearly.

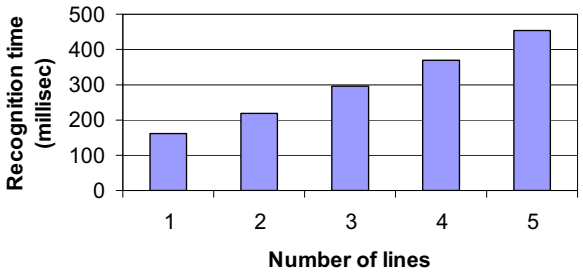


Figure 10. Impact of line count on recognition

We also found that for a given resolution, the recognition time depends primarily on the actual text content in an image. This is a more generic case (superset) of the above example wherein we varied the line count. Figure 11 shows the recognition time for various character pixels to total pixel ratio of a 5MP image.

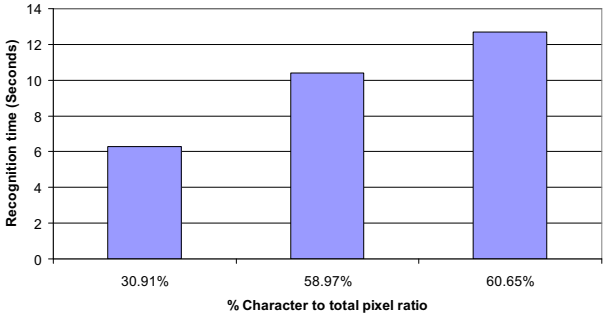


Figure 11. Impact of text content on recognition

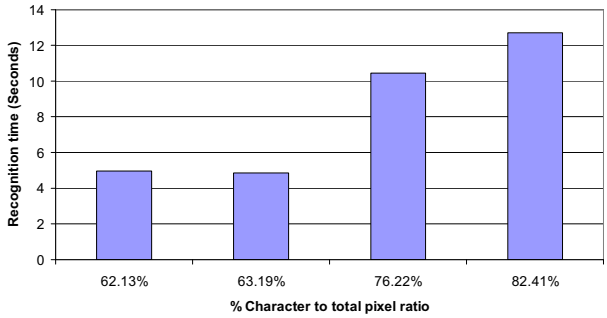


Figure 12. Recognition time for different text content ratio at various resolutions

To corroborate our earlier findings, we took the same input image at different resolutions and measured the recognition time. As shown in Figure 12, the recognition time greatly depends on character-total pixels ratio but does not increase linearly. Besides the line count and text content ratio, it is observed that recognition time can be affected by other factors such as clarity of image and sharpness. The investigation of these factors will be part of our future work.

3.2.2 Architectural Characteristics of OCRopus

Figure 13 shows the various architectural characteristics of OCR measured using Vtune[15] utility for different image resolutions. Figure 13(a) shows that the Cycles per Instruction (CPI) is high for all phases of OCR. This is due to the low throughput of in-order atom core.

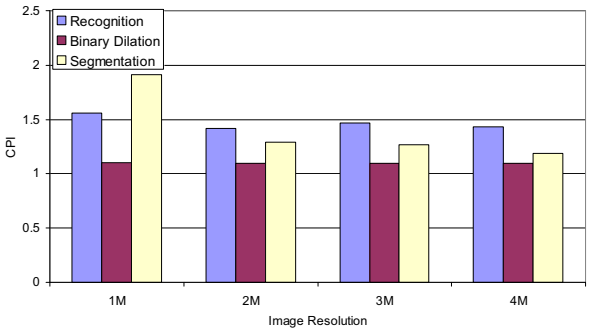


Figure 13 (a). CPI for OCR components

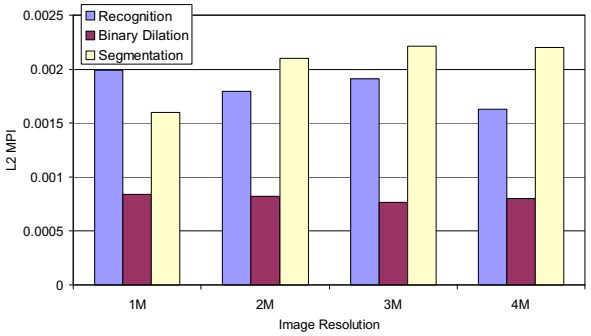


Figure 13 (b). L2 MPI for OCR Components

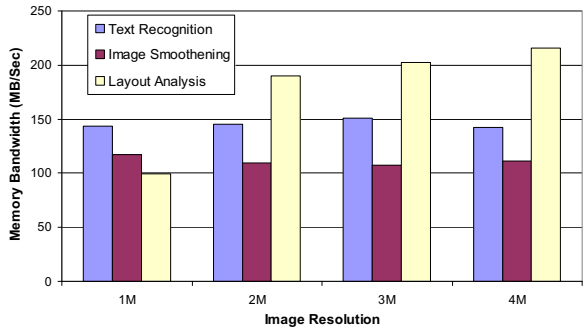


Figure 13 (c). DRAM Bandwidth

Figure 13. Architectural Characteristics of OCR Components

CPI remains constant for image smoothing and recognition phases for all image resolutions but increases significantly (by almost 50%) at 1MP for segmentation phase. This is due to the increased L1 data cache Misses per Instruction (MPI) for 1MP as shown in Figure 14(a). Figure 13(b) shows the L2 MPI and 13(c) shows the memory bandwidth for various phases. These graphs highlight the low L2 misses and shows the relatively low memory bandwidth utilization (~200MB/Sec which is less than 8% of the maximum throughput available in the platform).

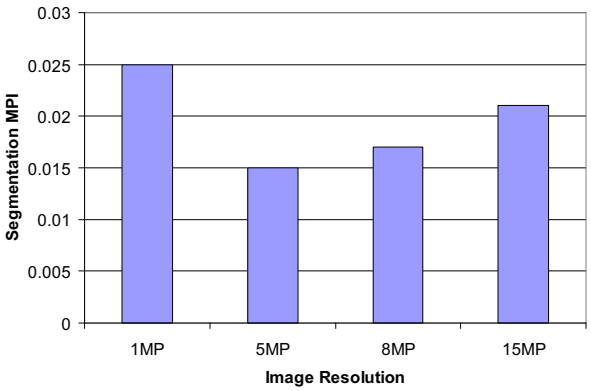


Figure 14 (a). L1 Data cache MPI for segmentation phase

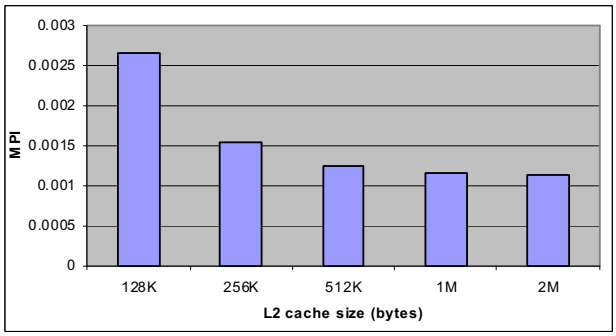


Figure 14 (b). OCR L2 MPI for various cache sizes

Figure 14 shows the various cache statistics obtained using CMPsim, a pin based cache simulator [16]. Figure 14(a) highlights the L1 MPI for segmentation phase. L1 MPI increases by almost 50% at 1MP, a significant degradation compared to 5MP, for the segmentation phase. This contributes to the low CPI in *segmentation* phase at 1MP as observed in 13(a). *Segmentation* phase has significant spatial locality. Hence, at higher resolution, prefetchers' effectiveness becomes pronounced and the L1 Data cache MPI is reduced.

Figure 14(b) shows the L2 MPI of the workload using CMPsim. We used a 5MP input image for this study and varied the L2 cache size from 512KB to 2MB with 8-way associativity. The results show that the working set size for 5MP image is around 1MB. L2 MPI varies from 0.0025 for 128KB cache to around 0.001 for 1MB cache. This shows that this workload is not memory bound and can fit in a small cache (512KB) as found in Atom platform. These results along with Vtune results corroborate that the application is computationally bound.

4. Software optimizations

As shown in previous sections, the execution time for text recognition is in the order of several seconds and is not appealing for real-time interactive usage. Further, the execution time increases with the image resolution and, our observation based on empirical analysis shows that we need 5 or 8 mega-pixel image resolution to achieve accurate recognition in a reasonable time. Moreover these resolutions are supported by the handheld devices as well [17][20].

In this section, we analyze various software optimizations for the various OCR phases to speedup its execution on Atom processor.

4.1 Image smoothing optimizations

As shown in previous sections, image smoothing takes a significant amount of time in OCRopus implementation especially for higher image resolutions. Hence *image smoothing*, which depends on the image resolution, is one of the main hotspot of OCR

Image smoothing is performed on each every pixel on the binarized image. Each new pixel value is computed independent of neighboring new pixel values. As there is no data dependency in this process, we start with multithreading mechanism. Figure 16 shows the various software optimization results at different resolutions for image smoothing phase.

1) Multi-threading (MT): We make use of multithreading capability (2 hardware threads) available in Atom. We threaded this function using p-thread libraries in Linux. We threaded the binary dilation across the various rows and columns and found the results to be similar. As shown, multithreading improves the performance by about 24%.

2) Computation Optimization (CO): Figure 15 shows the pseudo code for the base implementation of this function along with comments. The pixel for which binary dilation is computed is loaded twice for each comparison, as shown in line 6 and 7, which leads to 29 additional loads. Furthermore,

boundary conditions were checked for each pixel before the actual computation as shown in line 7. Therefore, we re-wrote the code to optimize the unnecessary computations. These miscellaneous compute optimizations (CO) yield significant reduction in runtime. Our results shows that CO alone improves the performance by over 3X.

```
1. For (i = -3; i <= 3; i++){  //radius along row
2.  for (j = -3; j <= 3; j++){   //radius along columns
3.     if (i*i+j*j <= 9){
4.      for(p=0; p<image.width;p++){ //stride through rows
5.         for(t=0; t<image.lenght;t++) //stride through columns
6.          new_image(p,t) =  //load pixel and compare
7.             max(image(p,t), image(check_border(p-i,t-j)))
8.         }
9.       }
   }
```

Figure 15. Pseudo code for binary dilation

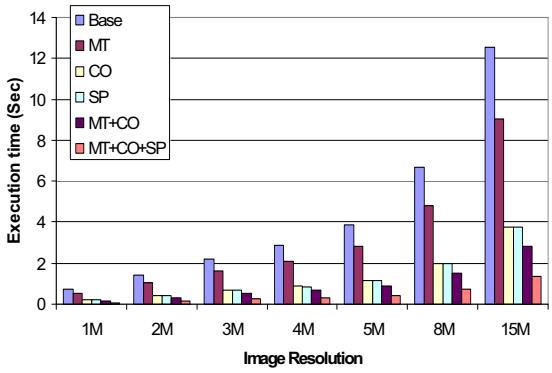


Figure 16. Image smoothing software optimizations

3) Sampling (SP): Sampling is a commonly used technique in image processing algorithms as a software optimization [21][22]. Instead of computing the maximum value for each pixel, we modified the algorithm to compute the maximal value for every other pixel. This effectively reduced the number of pixels computed by 75% as we skipped the even rows and columns. By sampling, we were able to reduce the execution time significantly while maintaining the recognition accuracy. Our results shows that sampling reduces the execution time by about 2.5X compared with the original code that has incorporated other optimizations.

We also combined the various optimizations (MT+CO), (MT+CO+SP) and results are shown in Figure 16. It can be observed that with all three optimizations, the execution time for binary dilation can be reduced by as much as 9X.

4.2 Segmentation optimization

After examining the code, we primarily applied multi-threading optimization to segmentation. The various functions in segmentation phase iterates over various pixels or bounding boxes (identified earlier on in the phase). Segmentation phase has high CPI due to the in-order nature of Atom. The independent instructions that would be able to fill the instruction pipeline on an out-of-order processor cannot be issued in an in-order core. By executing independent threads simultaneously, hyper-threading increases instruction issue width and helps to remove some of the pipeline bubbles. Figure 17 shows that multi-threading improve execution time by about 27% for various image resolutions. Every function in segmentation is benefited equally due to multi-threading as they all exhibit similar behavior.

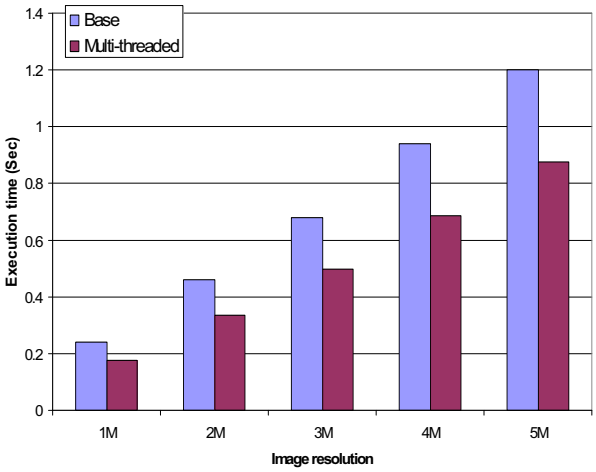


Figure 17. Multi-threaded segmentation

4.3 Recognition optimization

This phase has the highest CPI among the OCR components due to significant data dependency among instructions. Hence multi-threading provides the maximum benefit for this process. The multi-threading benefits for this phase are independent of the image resolution and text-content as the recognition of each character can be executed in parallel. Figure 18 (a) shows the multi-threading results for different text content in the image and, figure 18(b) shows it for various resolutions. We can observe that the execution time reduces by about 31% for this phase.

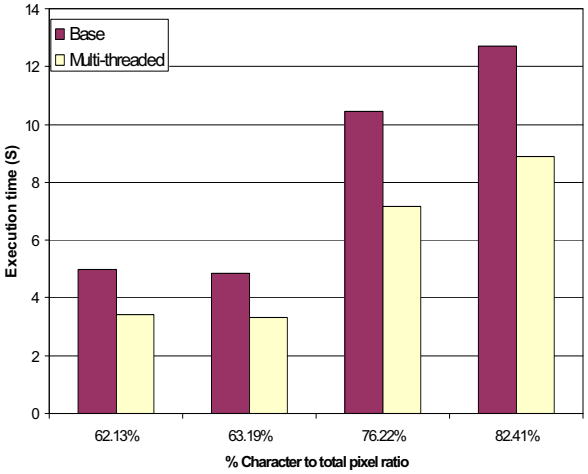


Figure 18 (a). Recognition speedup for different text content

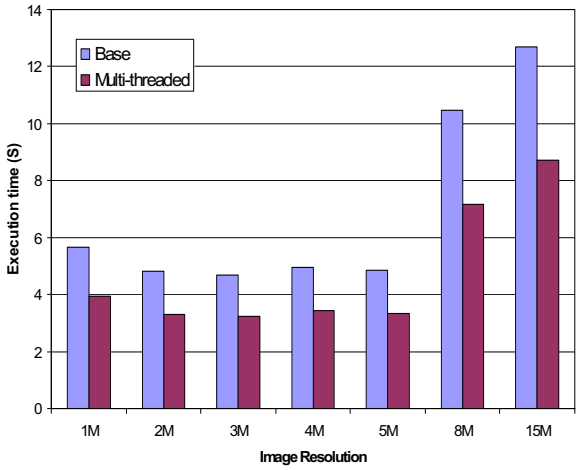


Figure 18(b). Multi-threaded Recognition for various resolutions

4.4 Software optimizations summary

Figure 19 summarizes all the software optimizations for various OCRopus phases. We reduced the overall execution time by at least 2X for a 5MP image and the *image* smoothing time (a hotspot) by almost 9X using various software and algorithm optimization techniques. The performance improvements are significant across all resolutions and increases with it.

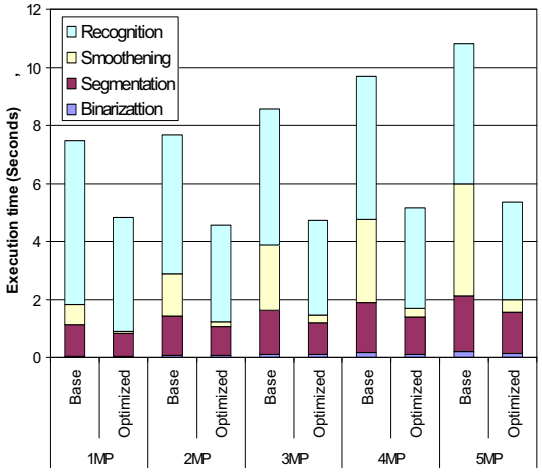


Figure 19. OCR Software optimizations

5. Hardware Acceleration

We designed and implemented a hardware accelerator to speedup the image smoothing process and optimize for power. A naïve approach is to calculate the maximum of the surrounding 29 values of current pixel (marked as red box in Figure 8), then move current pixel rightwards (and downwards to the leftmost pixel after a whole line is finished). However, in this naïve approach, each value is read many times from memory. A more efficient way is to read each pixel from memory just once and save the temporary max value of adjacent 5 and 7 values into registers for later use. The key component of this implementation called computing unit (CU) consists of two comparators, seven Row Registers (RR) and one control unit. As shown in Figure 20, one row (7 pixels) of the image is read from SRAM. The first comparator calculates three maximum values from 1, 5, and 7 pixels respectively and stores them in a 3-byte row register. Then the next row of 7 pixels is read and three maximum values are calculated and stored in the next row register, and so on. After reading and calculating seven rows of pixels, the seven row registers are filled up with maximum values for each row. Then the control unit selects one byte from each of the row registers and feed them into the second comparator. This comparator calculates the maximum value among the seven pixels, which is the final value for the new pixel. The new value can then be stored back to the SRAM.

Image smoothing process is performed on a binarized image, where the pixel values are 0 or 1 (indicating a black or white pixel). Hence the size of each pixel is a byte. Therefore, we replaced the comparators with OR logics to obtain the maximum value. This reduced the execution time and die area.

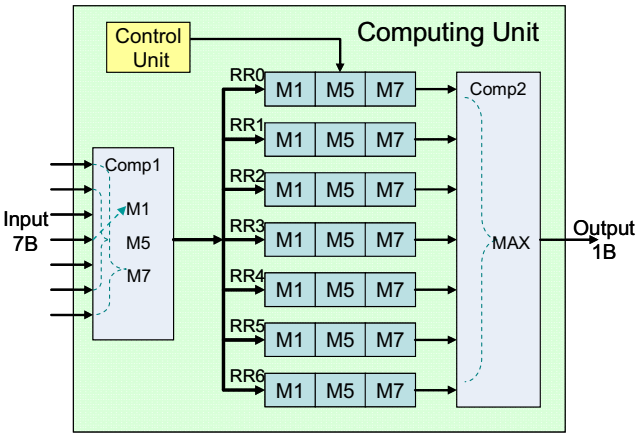


Figure 20. Overview of a Computing Unit

Note that the control unit controls which register row to fill for the next iteration and assures that the appropriate byte from each row is enabled. Essentially, each register row is read seven times before it is overwritten. It provides one byte in each cycle in the following order: M1, M5, M5, M7, M5, M5 and M1 (M1, M5, M7 represent the max value of adjacent 1, 5 and 7 pixels around current pixel). For instance, to calculate the first pixel, M1 from RR0 and RR6, M7 from RR3, and M5 from the rest of the row registers are chosen. When the next row is processed, RR0 is overwritten with the three new maximum values and, the next pixel is calculated using M1 from RR0 and RR1, M7 from RR4, and M5 from the rest of the row registers. An alternative way of controlling the row registers is to always store the three values of a processed row into RR0. Then each register is shifted to the next one in each cycle. For example, RR0 is shifted to RR1, RR1 to RR2, and so on. In this case, we always read M1 from RR0 and RR6, M7 from RR3, and M5 from the rest of the registers. However, the second approach incurs increased energy consumption. Therefore we adopted the first approach.

Once the first pixel value for the new image is obtained, the CU pipeline is filled up. As we keep reading and processing the next row in CU, we can get a column of new pixels at a speed of one pixel (byte) per CU cycle assuming that one row of seven pixels can be read from SRAM in one CU cycle. To get multiple columns of new pixels, we can use multiple CUs simultaneously. As shown in Figure 21, we can process M*N pixels using M CUs in (T + N) cycles where T is the initialization time before the pipeline is filled up. To speedup boundary pixel processing, we design a boundary control unit (BCU). BCU identifies boundary pixels and inserts padding pixels into the SRAM when the input image is loaded. Therefore, for an image size of X*Y pixels, the BCU will inserts 6X+6Y+36 padding pixels in total. If the SRAM can hold (X+6)*(Y+6) pixels, the new image with X*Y pixels can be obtained in X*(T+Y)/M cycles.

It is obvious that the performance can be improved with more CUs. However, this speedup is limited by the data transfer time (between memory and SRAM), which is limited by the main memory bandwidth. Our approach finds a reasonable M so that the next line is transferred to SRAM before it is being by CU, i.e., the computation time and the transfer time can be overlapped.

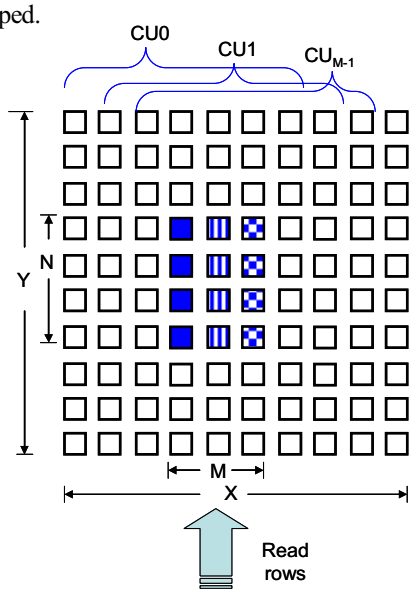


Figure 21. Using multiple CUs to parallel process columns

SRAM design in our accelerator is divided into input and output halves to hold the input and output image respectively. Let us assume an input SRAM with X*Y bytes. The data transfer efficiency increases with X as each byte is used seven times. In addition, the larger Y is, the computation efficiency we can achieve since the initialization time (to fill up the pipeline) can be better amortized. However since larger SRAM incurs more area, we choose a 128B by 256B (32KB) SRAM for both input and output for a total of 64KB. When the 64B output is ready, the data is transferred back to the main memory.

We implemented this accelerator using Xilinx 110T and runs at 250 MHz. Table 3 lists the detailed information about our implementation with 1 and 2 CUs along with 64KB SRAM. 1 and 2 CUs were chosen based on the sustainable memory bandwidth available in the platform which is about 2GB/s.

Table 3. Accelerator implementation using FPGA

CUs	1	2
Frequency	250 MHz	240 MHz
No. of Slice Registers:	710 of 69120 - 1%	1187 of 69120 - 1%
Block RAM/FIFO:	19 of 148 - 12%	19 of 148 - 12%
Dynamic Power	20.25(mW)	33.46(mW)

Figure 22 shows the execution time of image smoothing (in milliseconds) using our accelerator for various image resolutions. We can observe that for a 5MP image, the execution time with 2 CUs is about 12 ms, which is 33 times faster than our optimized code.

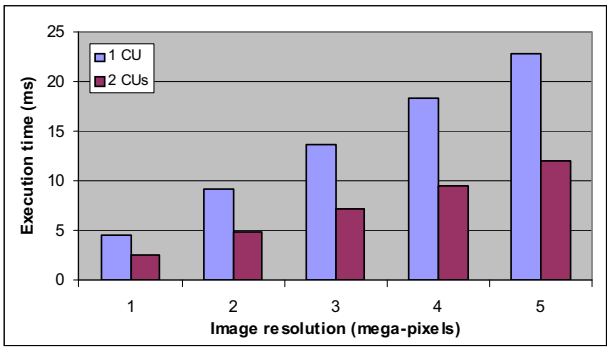


Figure 22. Execution time of image smoothing using 1 and 2 Computational Units with 64KB SRAM

We also synthesized our accelerator using 65nm technology. Table 4 lists the area and power consumption for 3 different frequencies. Each of these implementations gives a comparable performance to FPGA or better. We can observe that this implementation adds negligible overhead to a SoC platform in terms of power and area.

Table 4. Accelerator implementation using ASIC

	Freq(MHz)	Area(um^2)	Power(mW)
1CU	800	10656	4.527
	500	9811	2.739
	300	9753	1.641
2CU	800	14252	7.487
	500	13745	4.674
	300	13888	2.676

Figure 23 shows the final execution time of OCRopus with all the software and hardware optimizations. We can observe that the execution time has been reduced significantly from ~10 seconds to almost 4 seconds for 5MP image.

Figure 24 shows the energy-delay product of image smoothing process with the software and hardware optimizations in logarithmic scale. We measured the Atom's dynamic CPU power using power-meter to be 700mW and plotted the energy-

delay product for this phase compared to an FPGA implementation using 2 computation units. We can observe that energy-delay has been reduced by orders of magnitude for 5MP image compared to the base and software optimized code for this phase.

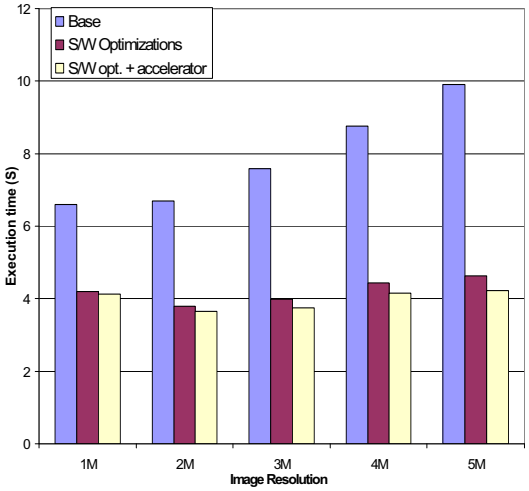


Figure 23. Total OCR Execution time with all optimizations

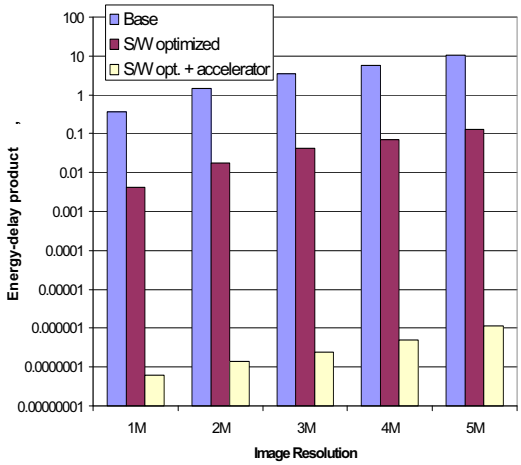


Figure 24. Energy delay for Image smoothing phase

6. Conclusions

In this paper, we analyzed the execution time of OCRopus processing on the Intel Atom CPU for handheld devices. We showed that the base software implementation requires more than 10 seconds for OCR processing even on a 1.6GHz core. We presented several software optimizations (multithreading, image sampling) as well as hardware acceleration to the OCRopus hotspots, implemented them and showed that these can improve the overall processing time by as much as 2X for a 5MP image and, almost an order of magnitude for a hotspot. We also described our hardware accelerator for image smoothing, a hotspot in OCR, which reduces the power consumption significantly.

As part of future work, we are looking into accelerator designs for other phases, segmentation and recognition, to further reduce execution time and, enable real time recognition in text to speech or other interactive applications. We also plan to enhance the software optimizations using vectorization available in the platform. We also would like to characterize the OCR performance with respect to precision for various inputs and characterize the application based on image clarity.

References

1. AFB, http://www.afb.org/AFBPress/pub.asp?DocID=aw090206
2. Quick-pen, http://www.quick-pen.com/index.php
3. Sun-Hwa Hahn, Joon Ho Lee, Jin-Hyung Kim, "A Study on Utilizing OCR Technology in Building Text Database", dexa, pp.582, 10th International Workshop on Database & Expert Systems Applications, 1999
4. S.V. Rice, F.R. Jenkins, T.A. Nartker, The Fourth Annual Test of OCR Accuracy, Technical Report 95-03, Information Science Research Institute, University of Nevada, Las Vegas, July 1995
5. R.W. Smith, The Extraction and Recognition of Text from Multimedia Document Images, PhD Thesis, University of Bristol, November 1987.
6. I. Marosi, "Industrial OCR approaches: architecture, algorithms and adaptation techniques", Document Recognition and Retrieval XIV, SPIE Jan 2007.
7. R. Smith, "An Overview of the Tesseract OCR Engine", Proceedings of the Ninth International Conference on Document Analysis and Recognition, 2007.
8. OCRopus, http://code.google.com/p/ocropus/
9. ABBYY, http://finereader.abbyy.com/
10. Expervision, http://www.expervision.com/tr7.htm
11. T. M. Breuel, "High performance document layout analysis," in Symposium on Document Image Understanding Technology, Greenbelt, MD, 2003.
12. "Openfst library." http://www.openfst.org/, 2007.
13. Ergina Kavallieratou, "A Binarization Algorithm specialized on Document Images and Photos", Proceedings of the Eighth International Conference on Document Analysis and Recognition, 2005
14. You Yang, "OCR Oriented Binarization Method of Document Image", Congress on Image and Signal Processing, 2008.
15. "Vtune Performance Analyzer", Intel Corporation, http://www.intel.com//software/products/vtune
16. Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce Jacob, "CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator", *Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2008.
17. Intel Reader, "http://www.intel.com/healthcare/reader/index.htm"
18. iPhone, http://www.apple.com/iphone/
19. iPad, http://www.apple.com/ipad/
20. Nokia N95, "http://www.nokiausa.com/find-products/phones/nokia-n95"
21. D.G. Lowe, Distinctive image features from scale-invariant keypoints, *International Journal of Computer Vision,* 60(2):91-110, 2004.
22. H. Bay, T. Tuytelaars, and L. Van Gool, SURF: Speeded up robust features, in *ECCV*, 2006.
23. Haidar Almohri, John S. Gray, Hisham Alnajjar, Real-time DSP-Based Optical Character Recognition System for Isolated Arabic characters using the TI TMS320C6416T, IAJC-IJME, 2008.

Benchmark Synthesis for Architecture and Compiler Exploration

Luk Van Ertvelde Lieven Eeckhout
Ghent University, Belgium

Abstract— **This paper presents a novel benchmark synthesis framework with three key features. First, it generates synthetic benchmarks in a high-level programming language (C in our case), in contrast to prior work in benchmark synthesis which generates synthetic benchmarks in assembly. Second, the synthetic benchmarks hide proprietary information from the original workloads they are built after. Hence, companies may want to distribute synthetic benchmark clones to third parties as proxies for their proprietary codes; third parties can then optimize the target system without having access to the original codes. Third, the synthetic benchmarks are shorter running than the original workloads they are modeled after, yet they are representative. In summary, the proposed framework generates small (thus quick to simulate) and representative benchmarks that can serve as proxies for other workloads without revealing proprietary information; and because the benchmarks are generated in a high-level programming language, they can be used to explore both the architecture and compiler spaces.**

The results obtained with our initial framework are promising. We demonstrate that we can generate synthetic proxy benchmarks for the MiBench benchmarks, and we show that they are representative across a range of machines with different instruction-set architectures, microarchitectures, and compilers and optimization levels, while being 30 times shorter running on average. We also verify using software plagiarism detection tools that the synthetic benchmark clones hide proprietary information from the original workloads.

I. Introduction

Benchmarking is at the foundation of architecture and compiler research and development. Computer architects and compiler designers make extensive use of benchmarks to evaluate their products and research ideas. Current benchmarking practice is to employ benchmarks that are derived from real-life applications. Although this is an effective approach, there are two major limitations. First, contemporary benchmarks have very large dynamic instruction counts, and as a result simulation is very time-consuming — it can easily take days or weeks to run a benchmark simulation to completion. Second, the available benchmarks may not be truly representative for real-life applications. In many cases, the real-life applications are proprietary, and companies are not willing to share their codes. Hence, third parties need to resort to (most often open-source) benchmarks that may not be truly representative for the real-life applications.

In this paper, we present a novel benchmark synthesis approach that aims at addressing these two limitations: the synthetic benchmarks are shorter than the original workloads they are modeled after (and hence they simulate faster), and they do not expose proprietary information (and can thus be distributed to third parties), yet they are representative with respect to real-life applications. The key novelty of the approach is that the synthetic benchmarks are generated in a high-level programming language, C in our case.

Our preliminary implementation and experimental evaluation using the MiBench benchmarks [1], multiple hardware platforms with very different instruction-set architectures (ISAs) and microarchitectures, and different compilers and optimization levels demonstrate the potential of the approach: the synthetic benchmarks mimick the real workloads well across architectures and compiler optimizations. We also provide evidence that a synthetic benchmark's source code does not provide any similarity with the original workload, hence it does not reveal proprietary information.

This work shares some commonalities with the recent line of research in statistical simulation [2]. The basic idea in statistical simulation is to collect a set of program characteristics by profiling a workload; these program characteristics are typically measured in the form of distributions. This statistical profile then serves as input for a synthetic workload generator. The synthetic workload then exhibits similar execution behavior as the original workload, so that it can be used as a proxy. Early proposals in statistical simulation generated synthetic traces [3], [4], [5]. While this is a reasonable approach for trace-driven simulation, it cannot be used on execution-driven simulators nor on real hardware. For that reason, researchers have proposed frameworks that generate synthetic benchmarks instead of synthetic traces [6], [7]. The synthetic benchmarks are generated at the binary level, hence they are tied to a particular ISA. Our work takes a significant step forward by generating synthetic benchmarks in a high-level programming language, which enables both architecture and compiler research. In addition, our framework models a program's control flow behavior more accurately than prior work and it generates synthetic code sequences using pattern recognition, not through statistics nor distributions of program characteristics.

More in particular, this paper makes the following contributions.

- We propose a framework and methodology for generating synthetic benchmarks in a high-level programming language that are representative for other workloads. Prior work generates synthetic benchmarks at the binary level which excludes using them across instruction-set architectures and compilers.
- We propose a novel structure, called the SFGL (Sta-

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

tistical Flow Graph with Loop information), to capture a program's control flow behavior in a statistical way. In particular, the SFGL captures a program's loop and basic block execution patterns. The SFGL enables the framework to generate function calls, (nested) loops and conditional control flow behavior in the synthetic benchmark. Prior work instead generated a linear sequence of basic blocks, but no loops nor function calls.

- We evaluate our framework on x86 and IA64 hardware. Prior work in synthetic benchmark generation was limited to evaluation through simulation, and/or considered RISC ISAs (Alpha, PowerPC) — RISC ISAs are easier to handle than CISC ISAs such as x86. We consider multiple hardware platforms with different ISAs, microarchitectures, compilers and optimization levels, and our preliminary results demonstrate good correspondence between the synthetic benchmarks and the original workloads in terms of performance sensitivity to the architecture, microarchitecture and compiler optimization level. In addition, the synthetic benchmarks are shown to be shorter running than the original workloads, and substantial simulation speedups are obtained.
- We demonstrate that the synthetic benchmarks do not reveal proprietary information. Two existing tools for identifying software plagiarism, Moss and JPlag, confirm that the synthetic benchmark does not show any similarity with the original workload.

II. Benchmark synthesis framework

The key problem to be solved in this paper is the following. We want to generate a synthetic benchmark in a high-level programming language that is similar to a real workload in terms of its execution behavior across architectures and compilers, yet it should not expose proprietary information and it should be short-running compared to the real workload. This is a non-trivial problem to solve. We now describe how we approach this problem at a high level — we will delve into the details later.

A. Framework overview

Figure 1 provides a high-level view of the overall framework. We start off from a real workload. This could be a proprietary application with a proprietary input. This workload is then compiled at a low optimization level, e.g., `-O0` in GNU's GCC. We then run the resulting binary and profile its execution, i.e., we count how often each function is called, how many times a loop is iterated, how often a branch is taken, how often a basic block is executed, etc. — this information is stored in the SFGL structure. In addition, we record memory access patterns as well as branch taken and transition rates. Finally, we employ a (simple) pattern recognizer that scans the executed code to identify C code statements that correspond to sequences of instructions observed at the binary level. This pattern recognizer translates the binary code to C code in a semi-random fashion in order to obfuscate proprietary information. All the characteristics that we collect are comprised

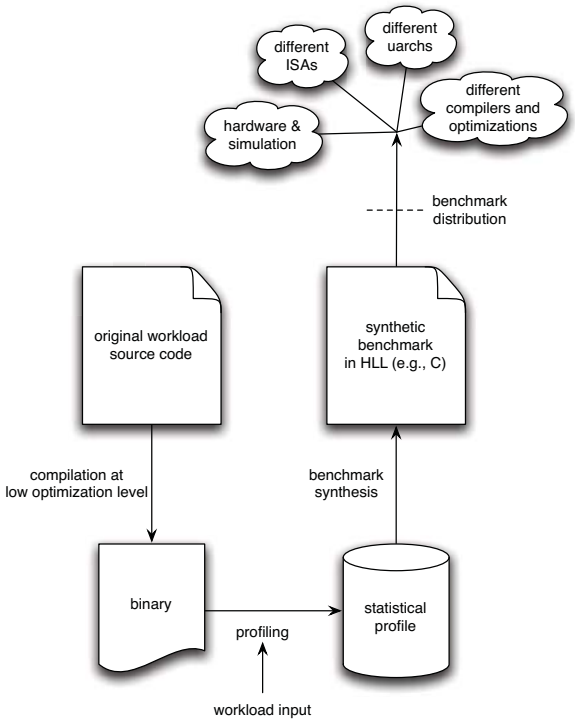


Fig. 1. Benchmark synthesis framework overview.

in a so called statistical profile that captures the behavior of the original workload and its input. We then generate a synthetic benchmark from this statistical profile. This is done in a high-level programming language (HLL), in our case C: we generate sequences of C code statements (basic blocks), as well as if-then-else statements, loops and function calls, and we add inter-statement dependencies as well as data memory access patterns. The C code structures are generated pro rata their occurrences in the original workload. However, we force the synthetic benchmark to execute fewer instructions than the original workload, by construction. This is done by reducing the execution frequencies of basic blocks, loops and function calls by a given reduction factor R. The end result is a synthetic benchmark that executes fewer instructions than the original workload while being representative for the original workload.

The synthetic benchmark does not expose proprietary information (because of the semi-random binary to source code translator, and the workload reduction) and can thus be distributed to third parties. Because the synthetic benchmarks are generated in a high-level programming language, they enable exploring both the architecture and compiler spaces, and compare systems with different compilers, and optimization levels, as well as different instruction-set architectures, microarchitectures and implementations. The synthetic benchmarks can run on execution-driven simulators as well as on real hardware.

An important aspect of our approach is that we compile the original workload at a low compiler optimization level before profiling. The reason for doing so is to force the compiler not

to perform aggressive optimizations. This facilitates the pattern recognition and translation from binary code to C code, and, more importantly, it enables generating synthetic benchmarks that can later be used to explore the compiler space, as we will demonstrate in this paper.

B. Applications

We believe this framework has a number of potential applications.

a) Distributing synthetic benchmarks as proxies for proprietary workloads: The most obvious application is to use the framework to generate synthetic clones for real-life proprietary workloads. There are many possible application scenarios, both in the embedded and server/datacenter spaces. For example, phone companies may not be willing to share their proprietary software with a processor vendor in order to optimize the processor architecture for the next-generation cell phone, yet they may be willing to share a synthetic clone. A similar application scenario applies to service providers in the cloud: they will be reluctant to share their platform software, yet they may want to distribute synthetic clones to third party hardware vendors. The same applies to compiler builders: they could evaluate their compiler performance based on the synthetic clones rather than the real workloads. Of course, co-optimization of hardware and software, which is an important focus today given the emphasis on energy-efficient computing, can also rely on synthetic benchmark clones.

The framework may also be an enabler for industry to share their workloads with their research partners in academia without revealing proprietary information. This will eventually lead to a more fruitful collaboration between industry and academia because the synthetic workloads may be more representative for the real-life commercial workloads than the open-source benchmarks used today.

b) Simulation time reduction: As mentioned earlier, the synthetic benchmarks are shorter running than the original workloads, i.e., their dynamic instruction count is significantly smaller. Because simulation time is an important concern in architecture research and development, benchmark synthesis also helps in reducing simulation time, and eventually the overall time-to-market. This is also important in the compiler space: for example, iterative compilation evaluates a very large number of compiler optimizations in order to find the optimum compiler optimizations for a given program [8], [9]. A synthetic clone that executes faster could reduce the overall compiler space exploration time.

c) Generate emerging workloads: The framework can also be used to generate emerging and future workloads. In particular, one can generate a statistical profile with performance characteristics that are to be expected for future emerging workloads. For example, one could generate specific sequences of C statements, a particular memory access behavior (e.g., large working set, random access patterns), etc. The synthetic benchmarks generated from these profiles can then be used to explore design alternatives for future computer systems.

d) Model hard-to-setup workloads: Similarly, one could build proxy benchmarks for workloads that are hard to setup. For example, database workloads and commercial workloads in general are non-trivial to setup [10]. Synthetics could be a way to facilitate the benchmarking process using commercial workloads. In fact, an additional advantage of generating synthetic benchmarks in a high-level programming language compared to assembly synthetic benchmarks is that interfacing libraries can be done easily using existing APIs. Although our current framework cannot be readily applied to mimicking commercial workloads, we believe it may be possible in future generations of our framework.

e) Benchmark consolidation: Multiple workloads can also be consolidated into a single synthetic benchmark. Basically, by putting together the statistical profiles from different workloads, one can generate a single consolidated synthetic benchmark that is representative for a set of workloads. Benchmark consolidation also helps hiding and obfuscating proprietary information.

III. Benchmark synthesis details

We now describe the details of our current framework. There are two major steps in the method: profiling a real workload and generating a synthetic benchmark clone.

A. Profiling

The profiling step can be implemented in a functional simulator or in a binary instrumenation tool, such as Pin [11] as we do in our framework. The profiler collects a number of characteristics.

1) Statistical Flow Graph with Loop annotation (SFGL): The central structure in the statistical profile is the SFGL which captures a program's control flow behavior in a statistical manner. Figure 2(a) shows an example. The nodes represent basic blocks and the edges represent control flow transitions. Each node is annotated with a basic block's execution frequency, and each edge is annotated with transition probabilities between nodes. The SFGL also identifies the loops along with the number of iterations that each loop executes.

For each instruction in each basic block we also record its instruction type. We consider a number of instruction types such as addition, subtraction, multiply, divide, etc., and we make a distinction between integer and floating-point instructions. We also keep track of the instruction's input operands (constant, register, memory) and output operand (register or memory).

2) Branch taken and transition rate: For each conditional branch that is not a loop back edge, we determine its taken and transition rate; the branch transition rate is defined as the number of times a branch changes between taken and not-taken during execution [12]. A low transition rate means that the branch is either mostly taken or mostly not taken, and a high transition rate means that the branch constantly changes between taken and not-taken. High and low transition rates typically suggest easy to predict branches. A medium

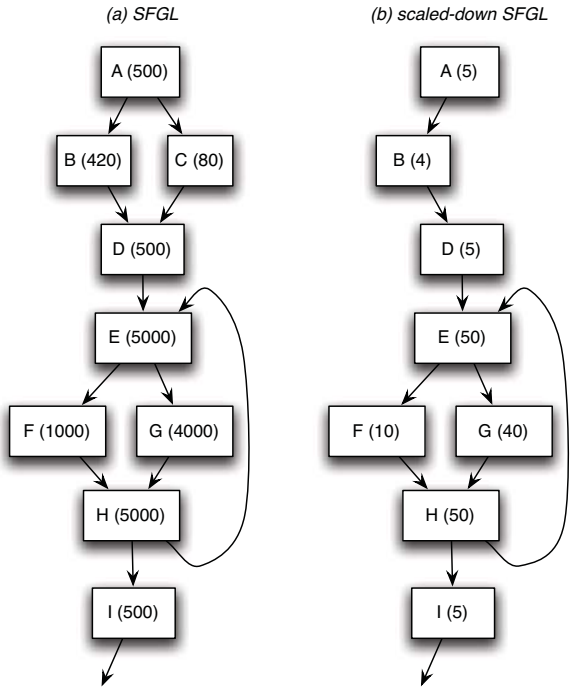


Fig. 2. An example SFGL: (a) computed SFGL and (b) scaled-down SFGL with a reduction factor $R = 100$.

class	*miss rate range*	*stride (bytes)*
0	0% - 6.25%	0
1	6.25% - 18.75%	4
2	18.75% - 31.25%	8
3	31.25% - 43.75%	12
4	43.75% - 56.25%	16
5	56.25% - 68.75%	20
6	68.75% - 81.25%	24
7	81.25% - 93.75%	28
8	93.75% - 100%	32

TABLE I

MEMORY ACCESS STRIDES FOR GENERATING A TARGET MISS RATE (ASSUMING A 32-BYTE CACHE LINE AND A 32-BIT ARCHITECTURE).

transition rate suggests hard to predict branches. The branch transition rate is independent of a particular branch predictor, and we classify branches into two classes, easy to predict branches (either high or low branch transition rate) and hard to predict branches.

3) Memory access patterns: For each memory access we record its cache hit/miss ratio. We do this by simulating a cache structure during profiling — there exist tools that compute cache miss rates across a range of cache organizations in a single pass [13]. We classify the memory accesses in a number of classes according to their hit/miss ratios. We identify 8 classes for different ranges of miss rates, see Table I; we will use these classes to generate specific memory access patterns, as we will describe later.

B. Synthetic benchmark generation

The second step in our framework is to generate a synthetic benchmark from the statistical profile. This is done in a number of steps.

1) Scale-down SFGL: We first compute a scaled-down SFGL by reducing the occurrences of the basic block and loop counts in the SFGL. This is done by dividing the basic block execution counts and loop iteration counts by a reduction factor R. For nested loops, we first scale the iteration count of the outer loop. If the iteration count of the outer loop is smaller than the reduction factor, we also downscale the nested loop, etc. Basic blocks and loops that are executed infrequently (i.e., less than R times) are removed from the SFGL. The purpose for downscaling is to generate short-running synthetic benchmarks, and obfuscate the original workload's semantics. Figure 2(b) shows the downscaled SFGL of the example shown in Figure 2(a). The reduction factor equals 100 in this example; basic block C does no longer appear in the down-scaled SFGL.

2) Generate basic blocks and loops: We now start generating the skeleton for the synthetic benchmark. We pick a random basic block based on the (reduced) execution counts, i.e., a basic block with a large execution count has a higher probability for being selected than a basic block with a lower execution count. If this basic block is part of a loop, we generate the loop that contains this basic block; for example, if basic block F would be picked in Figure 2(b), the framework would generate a loop comprising basic blocks E, F, G and H. If the loop itself is nested in a bigger loop, we first generate the outer loop and then generate the inner loops. If the basic block is not part of a loop, we determine its successor(s) and start building the control flow structure of the synthetic benchmark. If there are no successors to a basic block (because the successor basic blocks got removed during down-scaling), we re-start the generation algorithm and pick a random basic block. For each basic block and loop that we generate, we decrease the respective execution counts to reflect the fact that these basic blocks and loops have been generated. Basic blocks and loops with zero execution counts are removed from the SFGL. We continue this process until all basic blocks in the SFGL have been selected and the SFGL is empty.

3) Function assignment: We subsequently organize the basic blocks and loops to functions. This organization does not necessarily correspond to the functions observed in the original workload — again, this is to hide proprietary information in the synthetic benchmark.

4) Generate C statements: Once we have the skeleton synthetic benchmark consisting of functions, loops and basic blocks, we now populate the basic blocks with C statements. This is done by scanning the instruction types of all the instructions in each basic block, and by identifying C statements that correspond to these sequences of instructions. Table II shows the most important patterns and how they are translated into C statements. These patterns cover over 95% of the dynamic instructions for all the benchmarks. Coverage is not 100% (which again helps hiding proprietary information): to compensate for the uncovered instructions we keep track of the number of operations and types that have been translated so far, and we compensate for those instructions on a later occasion. For example, if we are lagging behind in the number

pattern	*example*	*C statement*
load-store	`movl t+512, %eax` `movl %eax, t+504`	`mem[i] = mem[j];`

load-arithmetic-store	`movl t+512, %eax`
	`addl $2,%eax`
	`movl %eax, t+504`
		`mem[i] = mem[j] op cst;`

load-load-arith-store	`movl t+508, %edx`
	`movl t+512, %eax`
	`leal (%edx,%eax), %eax`
	`movl %eax, t+504`
		`mem[i] = mem[j] op mem[k];`

load-load-arith-load-reg-arith-reg-store	`movl t+508, %edx`
	`movl t+512, %eax`
	`addl %eax, %edx`
	`movl t+516, %eax`
	`movl %edx, %ecx`
	`subl %eax, %ecx`
	`movl %ecx, %eax`
	`movl %eax, t+504`
		`mem[i] = mem[j] op mem[k] op mem[l];`

load-cmp-br	`movl t+504, %eax`
	`cmpl $3, %eax`
	`jbe`
		`if (mem[i] > cst)`

store	`movl $9, %eax`	`mem[i] = cst;`

TABLE II

GENERATING C STATEMENTS THROUGH PATTERN RECOGNITION. THE `op` REFERS TO AN OPERATION (E.G., ADDITION, SUBTRACTION, ETC.); THE `cst` REFERS TO A RANDOMLY GENERATED CONSTANT VALUE.

of loads, we try to generate a 'load-load-arith-store' pattern instead of a 'load-arith-store' pattern. Or, if we are lagging behind in the number of stores, we will generate an additional 'store' pattern.

When reaching the end of a basic block we generate a branch statement. This can be either a loop back edge or a conditional branch. In case of a loop back edge, we generate a `for` loop with the iteration count the number of times the loop needs to be iterated according to the scaled-down SFGL. For a non-loop branch, we generate an if-then-else statement, and we make a distinction between easy to predict branches and hard to predict branches. The easy to predict branches are assumed to be either always taken or always not-taken. The non-executed path is filled with C statements that print out the results that have been computed elsewhere in the synthetic benchmark — this is to force the compiler not to optimize code away that is needed to preserve representativeness while producing data that is never used. The hard to predict branches jump in one or the other direction based on their transition rate using a modulo operation on a loop iterator. For example, a conditional branch with a transition rate of 30% is modeled using an operation that computes modulo 3 on the iterator of its innermost outerloop.

Finally, we also generate memory access patterns. This is done by generating stride patterns for all memory accesses, following prior work by Joshi et al. [7] who found that over 90% of the memory references can be modeled as stride access patterns. These patterns walk through pre-allocated memory with a particular stride; the stride value is determined by the memory access' hit/miss ratio. Different hit/miss ratios lead to different stride values. For example, an always hit memory access is modeled through a zero stride. A 50% hit rate is modeled through a stride value of 4; this will lead to a 50% miss rate assuming a 32 byte cache line size and a 32-bit machine, see also Table I.

C. Example

Figure 3 shows an example for the fibonacci kernel: it shows the original code along with the automatically generated synthetic clone. The profiling was done with a particular input; this is reflected in the number of iterations that the loop is taking: the synthetic benchmark takes 20 iterations while this is an input parameter in the original program. That specific input never caused an overflow, hence the if-statement in the loop is never executed. This example illustrates that the fibonacci kernel is no longer recognizable in the synthetic clone because the data dependencies between the statements are different between the original program and its clone.

D. Limitations

The current framework has a number of limitations which we plan to address as part of our future work.

- Different program characteristics are modeled independently of each other — the framework currently takes a first-order approach and assumes that the characteristics are uncorrelated. For example, memory access behavior is modeled independently of control flow behavior and its interaction is not modeled. This is obviously not the case in real programs. Modeling second-order effects is likely to improve accuracy.
- The memory access behavior is based on cache miss rates and hence it is specific to a particular memory hierarchy. Although it is possible to measure cache miss rates for a range of caches in a single run, as mentioned before, a better solution would be to have a microarchitecture-independent way of modeling memory access behavior.

(a) original program

```
int fib (int n) {
  int a=0, b=1, i, sum=0;

  for (i=0; i<n; i++) {
    sum=a+b;
    if (sum<0) {printf("overflow"); break;}
    a=b;
    b=sum;
  }
  return sum;
}
```

(b) synthetic program clone

```
unsigned int mStream0[256];
int i,j;

int f () {
  for (i=0; i<20; i++) {
    mStream0[4]=mStream0[7] + mStream0[2];
    if (mStream0[0]==0x99) {
      for(j=0;j<256;j++)
         printf("%d;", mStream0[j]);
      }
    mStream0[6]=i;
    mStream0[7]=mStream0[6];
  }
}
```

Fig. 3. The original fibonacci kernel (a) and its synthetic clone (b).

Further, the current approach assumes that memory accesses can be modeled using stride patterns, which is a reasonable approach according to [7], as discussed above. Future work though may focus on modeling less regular memory access patterns.

- The ILP model is simplistic in our current setup, as we assume random dependencies between instructions. A more accurate approach would be based on profiling information so that the distribution of data dependencies in the synthetic matches the original workload. A downside of making the approach more accurate though is that it may reveal proprietary information — there is a trade-off in accuracy versus hiding proprietary information.
- The reduction factor is chosen empirically so that the synthetic benchmark executes approximately 10 million instructions, as we will describe later. A more accurate approach would base the reduction factor on how representative the synthetic workload is relative to the real workload.

IV. Experimental setup

We use the MiBench benchmark suite [1] in our experimental setup, which is representative for the embedded market. We use MiBench in this work because the embedded space is a target application domain for the framework proposed in this paper, cf. the use case mentioned before in which a cell phone company may be willing to distribute a synthetic benchmark to hardware vendors as a proxy for its proprietary software. Our initial results with these benchmarks are promising, as we will describe in the next section. We will consider more complicated workloads as part of our future work.

machine	*ISA*	*description*
Pentium 4, 3GHz	x86	Pentium 4 at 3GHz w/ 1MB L2
Core 2	x86_64	Core 2 at 2.2GHz w/ 2MB L2
Pentium 4, 2.8GHz	x86	Pentium 4 at 2.8GHz w/ 1MB L2
Itanium 2	IA64	Itanium 2 at 900MHz w/ 256KB L2
Core i7	x86_64	Core i7 at 2.67GHz w/ 8MB L2

TABLE III
MACHINES USED IN THIS STUDY.

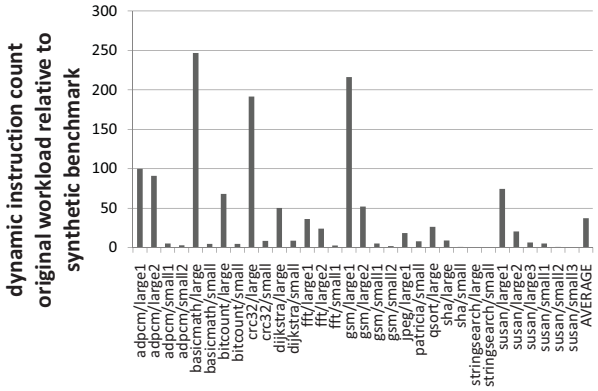


Fig. 4. Reduction in dynamic instruction count.

The profiling is done using Pin [11], which is a dynamic binary instrumentation tool. The cache simulations are done using Pin as well. The branch prediction results are obtained using PTLSim [14]; we consider a hybrid branch predictor with a bimodal component along with a history-based component. We also run detailed cycle-accurate simulations using PTLSim and we simulate a 2-wide out-of-order processor.

We also run real hardware experiments on five machines, see Table III. The machines include Pentium 4, Core 2, Core i7 and Itanium 2 processors, and three ISAs: x86, x86_64 and IA64.

We use GNU's GCC compiler v4.0.2 in all of our experiments for the x86 and x86_64 machines; we use GCC v3.3.2 on the Itanium 2 machine. We consider four compiler optimization levels: `-O0`, `-O1`, `-O2` and `-O3`.

V. Evaluation

The evaluation of the framework is done in a number of steps: we evaluate whether the synthetic benchmarks correspond to the real workloads with respect to their dynamic instruction count, instruction mix, cache performance, branch prediction behavior, and eventually overall performance across architectures and compilers.

A. Dynamic instruction counts

Figure 4 shows the reduction in dynamic instruction count between the synthetic benchmark and the original workload.

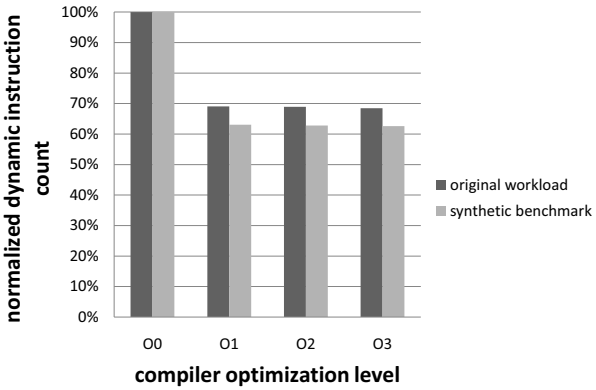


Fig. 5. Normalized dynamic instruction count across compiler optimization levels.

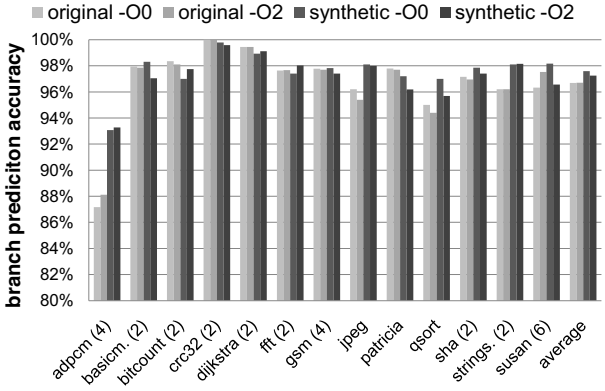


Fig. 9. Branch prediction rates for the original workloads and the synthetic benchmarks.

Recall that the synthetic benchmark generation process scales down the SFGL using a reduction factor. We choose the reduction factor such that the synthetic benchmark executes approximately 10 million instructions. This leads to a reduction factor ranging from 1 to 250. The fact that the reduction factor is low in a number of cases is due to the fact that some MiBench benchmarks are fairly short running, hence there is little simulation time reduction to be gained. On average though, we achieve a 30× reduction in dynamic instruction count.

Figure 5 shows the normalized dynamic instruction count across compiler optimization levels; we show average numbers here. The dynamic instruction count is an important optimization target for compilers, even on today's superscalar out-of-order processors [15]. The synthetic workload tracks the original workload fairly well: both suggest that the dynamic instruction count reduces by about a third when going from `-O0` to a higher optimization level.

B. Instruction mix, cache, and branch prediction behavior

Figure 6 shows the instruction mix at the `-O0` and `-O2` optimization levels. Figures 7 and 8 show similar graphs for the data cache behavior (we consider fairly small cache sizes in order to stress our framework); and Figure 9 shows results for the branch predictor behavior. All of these graphs basically lead to the same conclusion. Although the synthetic benchmarks do not yield a perfect match with the original workload, they most often yield the same conclusions and insights. For example, both the synthetics and the real workloads see a decrease in the fraction of load instructions along with an increase in the fraction of arithmetic instructions at a higher optimization level, see the average bars on the righthand side in Figure 6(a) and (b); the reason is that optimizations such as copy propagation eliminate load instructions. In terms of data cache behavior, `dijkstra` seems to be the benchmark that is most sensitive to cache space, see Figure 7(a), and a data cache size of 8KB seems to capture most of the benchmark's working set (i.e., there is a significant increase in data cache hit rate going from 4KB to 8KB but a minor increase going from 8KB to 16KB). We observe the same trend for the synthetic version of `dijkstra`, see Figure 7(b). Finally, for the branch predictor accuracy graph, see Figure 9, we observe that `adpcm` is most sensitive to the branch predictor; this is also captured by the synthetic workload.

C. Detailed cycle-accurate simulation

Figure 10 shows CPI for a 2-wide out-of-order processor while varying cache size; these results are obtained through detailed cycle-accurate simulation using PTLSim. The synthetics track fairly well overall performance across the benchmarks. For example, `fft` is the benchmark with the highest CPI (due to a large fraction of floating-point instructions) and `sha` the lowest CPI; we observe this for both the real and synthetic workloads. We also observe that the synthetic workload captures the performance trend as a function of data cache size well, see for example `dijkstra` and `qsort`. The remaining errors come from a number of potential sources. The current data dependency model can be improved to more accurately mimick real application behavior in the synthetic benchmarks (see `bitcount`); also, modeling the branch behavior can be improved upon (see `adpcm`), as well as the data cache behavior (see `stringsearch`). Improving the modeling of these program characteristics is likely to improve the representativeness of the synthetic benchmarks compared to the original workloads.

D. Overall performance across architectures and compilers

Figure 11 shows the normalized execution times for the original workloads and the synthetic clones across different architectures and compilers and optimization levels; we consider the real machines and a benchmark consolidation setup here and report average numbers. All the results are normalized to the `-O0` optimization level on the Pentium 4 3GHz machine. This graph shows that the synthetic workloads track the original workloads fairly well across architectures and compiler optimization levels. The error in predicting the speedup relative to `-O0` is less than 20% across all machines and optimization levels, with an average error of 7.4%. The synthetics track that the Core i7 yields the best overall performance, and the Itanium 2 the worst. A particularly encouraging

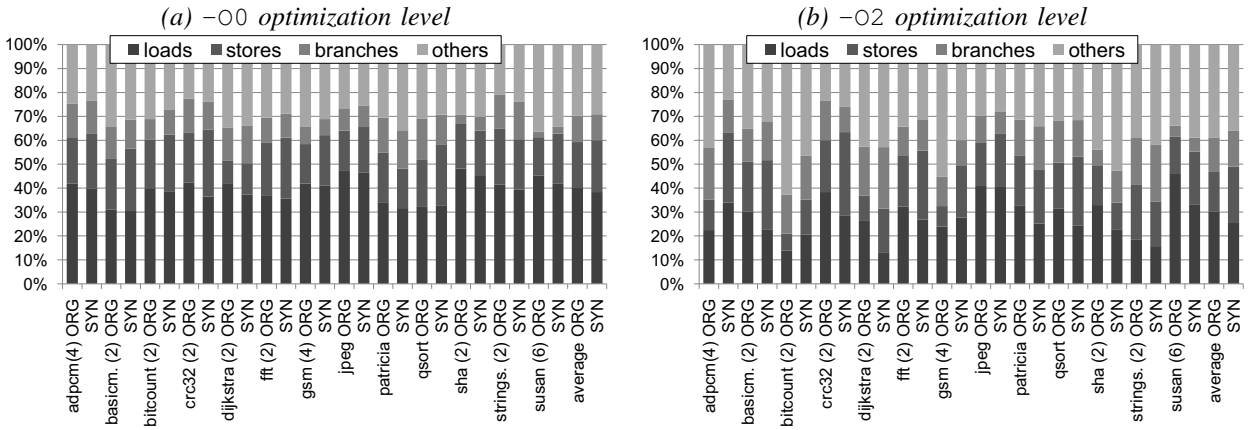


Fig. 6. Instruction mix for (a) the `-O0` optimization level and (b) the `-O2` optimization level.

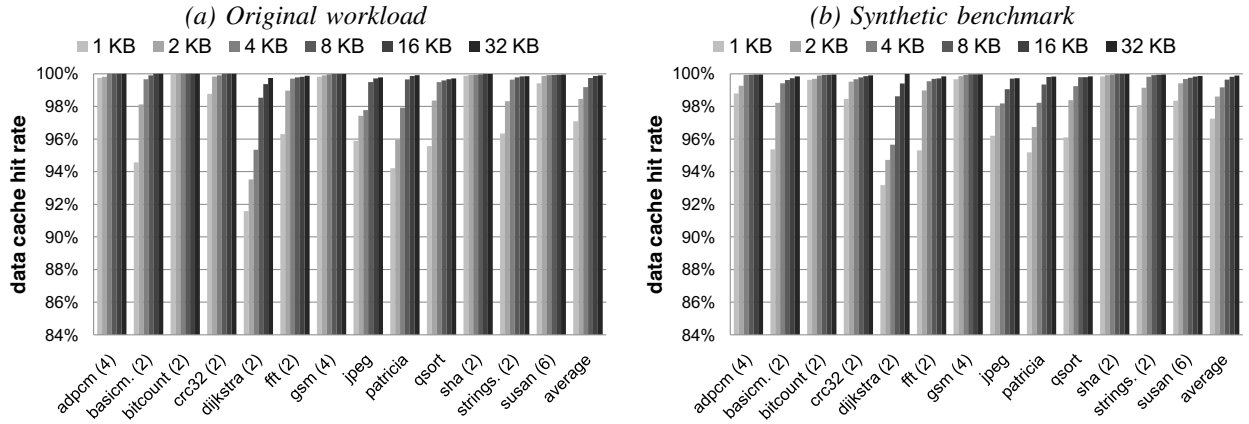


Fig. 7. Data cache hit rates for (a) the original workloads and (b) the synthetic benchmarks at the `-O0` optimization level.

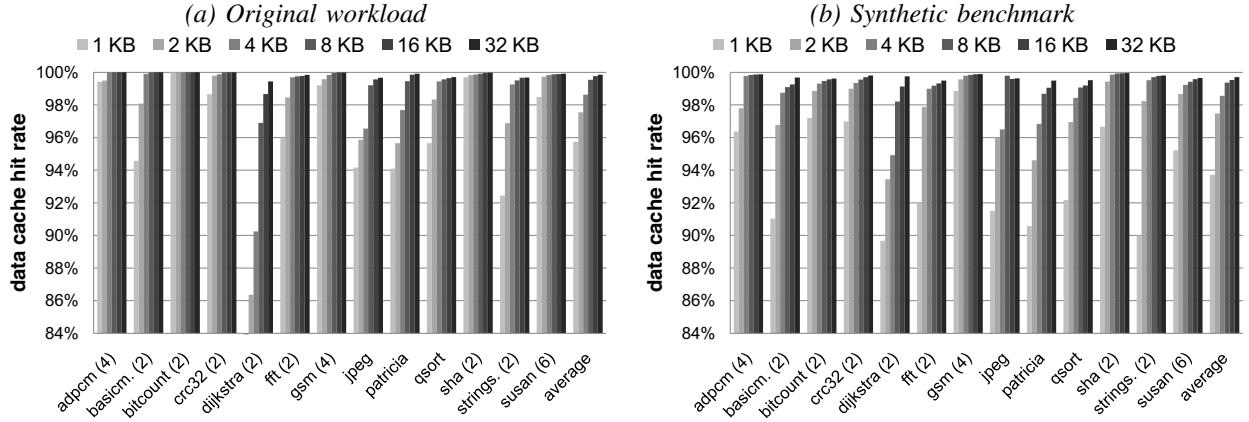


Fig. 8. Data cache hit rates for (a) the original workloads and (b) the synthetic benchmarks at the `-O2` optimization level.

result is that the synthetic workload is able to track that the `-O2` and `-O3` optimization levels yield a substantial 25% performance benefit over `-O1` on the Itanium 2 machine but not on the other machines. This performance benefit for the Itanium architecture is due to the fact that Itanium's EPIC architecture is sensitive to compiler optimizations — an EPIC architecture is a statistically scheduled architecture as opposed to a dynamically scheduled out-of-order processor, hence compiler optimizations may have a more significant impact on overall performance. Clearly, the synthetic workloads expose program constructs similar to the real workloads that enable the compiler to optimize in a similar vein.

E. Benchmark obfuscation

An important asset of benchmark synthesis is that it hides proprietary information, i.e., it is impossible, or at least very hard, to reverse engineer proprietary information from the synthetic benchmark. One way of evaluating whether this is really achieved is through manual inspection. By comparing

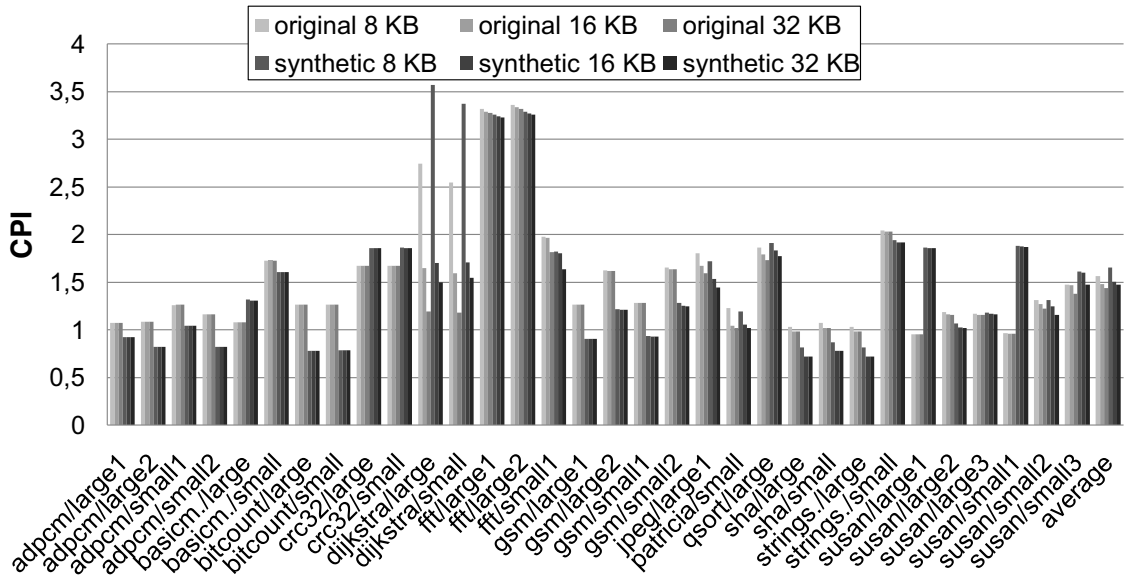


Fig. 10. CPI for the original and synthetic workloads on a 2-wide out-of-order processor while varying cache size.

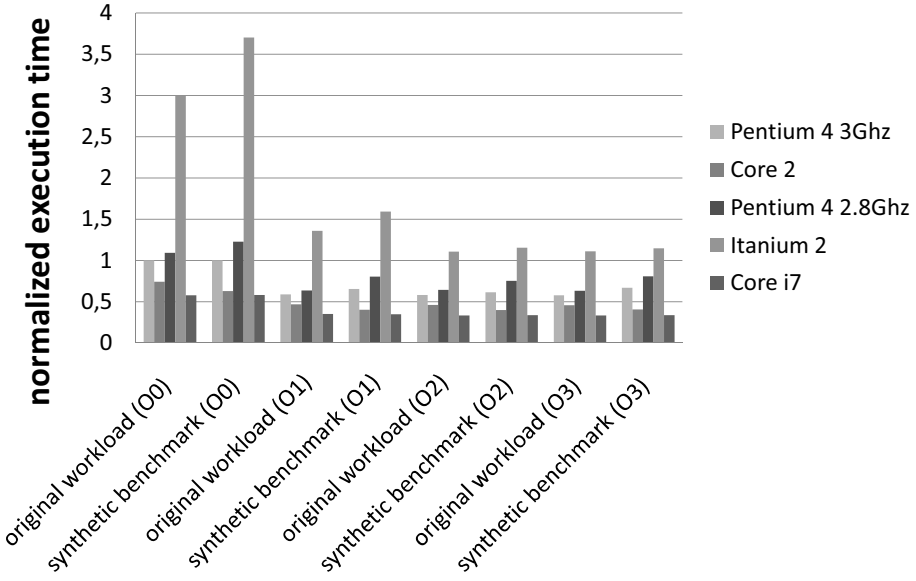


Fig. 11. Normalized average execution time for the original workloads and synthetic clones across architectures and compilers.

the synthetic benchmark against the original workload, one can assess whether any proprietary information is still left in the synthetic benchmark. Presumably, companies that plan on using benchmark synthesis will most likely do this validation process very carefully before distributing a synthetic clone.

We use existing tools for evaluating whether proprietary information is still present in the synthetic benchmark. We therefore use two existing tools, Moss [16] and JPlag [17], which are used to find plagiarism in software. Moss' main usage has been in detecting plagiarism in programming classes; JPlag is aware of programming language syntax and program structure. The way both tools work is that the user gives two source code files, and the tool returns whether there is any similarity between these two files. When giving the original workload and the synthetic benchmark, both Moss and JPlag return that the synthetic benchmark does not provide any similarity with the original workload.

VI. Related work

As mentioned in the introduction, this work shares some commonalities with statistical simulation. Statistical simulation [3], [4], [5] collects program characteristics from a program execution and subsequently generates a synthetic trace from it which is then simulated on a simple, statistical trace-driven processor simulator. The important advantage of statistical simulation is that the dynamic instruction count of a synthetic trace is very short, typically a few millions of instructions at most, making it a useful simulation speedup

technique for quickly identifying a region of interest in a large microprocessor design space. A synthetic trace hides proprietary information very well, however, a synthetic trace cannot be run on real hardware nor on an execution-driven simulator (which is current practice as opposed to trace-driven simulation).

Synthetic benchmarks such as Whetstone [18] and Dhrystone [19] are manually crafted benchmarks. Manually building benchmarks though is both tedious and time-consuming, and in addition, these benchmarks are quickly outdated. Therefore, recent work proposed automated synthetic benchmark generation [6], [20], [7], [21] which builds on the statistical simulation approach but generates a synthetic benchmark rather than a synthetic trace. Van Ertvelde and Eeckhout [22] proposed code mutation which is a different approach to hiding proprietary information. They mutate an existing benchmark so that reverse engineering the benchmark gets more complicated, while preserving similar execution behavior. The advantage of these approaches compared to the original proposals in statistical simulation is that the synthetic benchmarks can be executed on real hardware as well as on execution-driven simulators. However, the benchmarks are generated at the binary level, hence they can be used to drive architecture exploration only; they cannot be used for compiler research and development, nor can they be used for hardware/software co-optimization.

Sampled simulation [23], [24], [25], [26] selects a number of representative samples from the dynamic instruction stream. Simulating these samples instead of the complete dynamic instruction stream yields simulation speedups of several orders of magnitude. Although having only samples to analyze will complicate the understanding the functional semantics of the proprietary application, it may still reveal sensitive information, i.e., if the sampled trace is representative for the entire program execution, it will most likely reveal proprietary information.

Our framework borrows some concepts proposed in prior work. Our proposal extends the statistical flow graph (SFG) [3] with loop information. By doing so, our synthetic benchmarks consist of many (nested) loops as observed in real workloads. Prior work in benchmark synthesis [6] on the other hand, generates a linear sequence of instructions that is iterated in a big loop until convergence. Our framework also borrows the idea of using the branch transition rate for modeling the branch behavior [7] and the stride-based memory access pattern modeling approach [6]. The main difference with this prior work though is that (i) we target synthetic benchmarks in a high-level programming language, whereas prior frameworks generated synthetic traces or benchmarks in assembly, (ii) we generate fine-grained loop structures using the SFGL, and (iii) we use pattern recognition rather than statistics to generate synthetic code sequences.

Code obfuscation [27] converts a program into an equivalent program that is more difficult to understand and reverse engineer. There is a fundamental difference between code obfuscation and benchmark synthesis though. The goal of program obfuscation is to generate a transformed program that is functionally equivalent to the original program, i.e., when given the same input, the transformed program should produce the same output as the original program. The performance characteristics of the transformed program can be — and in practice they are — very different from the original program. Benchmark synthesis on the other hand generates a synthetic program that exhibits the same performance characteristics as the original program, however, its functionality can be very different. Not having to preserve functionality has an important implication for benchmark synthesis because it allows for generating a synthetic benchmark for a specific input, hence benchmark synthesis can also hide proprietary information as part of the input.

VII. Conclusion and future work

This paper proposed a novel benchmark synthesis paradigm that generates synthetic benchmarks in a high-level programming language. It generates small but representative benchmarks that can serve as proxies for other workloads without revealing proprietary information; and because the benchmarks are generated in a high-level programming language, they can be used to explore the architecture and compiler space. Our experimental results are promising and demonstrate the feasibility and effectiveness of the approach. We demonstrate good correspondence between the synthetic and original workloads across instruction-set architectures, microarchitectures and compiler optimizations. This paradigm has many potential applications: distribution of proprietary applications as proxies, drive architecture and compiler research and development, speed up simulation, model emerging and hard-to-setup workloads, and benchmark consolidation.

While the results are promising, there is ample room for improvement and future work. There are various aspects that could be modeled more accurately, such as modeling data dependencies following dependence patterns seen in the original workload, modeling the memory access patterns in a microarchitecture-independent way, etc. However, increasing the representativeness and similarity of the synthetic benchmark with respect to the original workload in terms of its execution behavior, may lead to revealing some proprietary information — there is a trade-off in representativeness versus hiding proprietary information. Being able to generate multi-threaded workloads is another obvious extension to our framework. Subsequently, extending the framework towards more complex workloads, i.e., commercial workloads, is yet another avenue of future work: this will enable the framework to be used in the high-end server market segment as well.

Acknowledgments

We would like the reviewers for their thoughtful comments and suggestions. This research is supported in part by the FWO projects G.0232.06, G.0255.08, and G.0179.10, and the UGent-BOF projects 01J14407 and 01Z04109.

REFERENCES

[1] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC)*, Dec. 2001.

[2] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere, "Statistical simulation: Adding efficiency to the computer designer's toolbox," *IEEE Micro*, vol. 23, no. 5, pp. 26–38, Sept/Oct 2003.

[3] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John, "Control flow modeling in statistical simulation for accurate and efficient processor design studies," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, June 2004, pp. 350–361.

[4] S. Nussbaum and J. E. Smith, "Modeling superscalar processors via statistical simulation," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2001, pp. 15–24.

[5] M. Oskin, F. T. Chong, and M. Farrens, "HLS: Combining statistical and symbolic simulation to guide microprocessor design," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, June 2000, pp. 71–82.

[6] R. Bell, Jr. and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *Proceedings of the 19th ACM International Conference on Supercomputing (ICS)*, June 2005, pp. 111–120.

[7] A. M. Joshi, L. Eeckhout, R. Bell, Jr., and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, no. 2, Aug. 2008.

[8] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, May 1999, pp. 1–9.

[9] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast searches for effective optimization phase sequences," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2004, pp. 171–182.

[10] M. Shao, A. Ailamaki, and B. Falsafi, "DBmbench: Fast and accurate database workload representation on modern microarchitecture," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, Oct. 2005, pp. 254–267.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, June 2005, pp. 190–200.

[12] M. Huangs, P. Sallee, and M. Farrens, "Branch transition rate: A new metric for improved branch claassification analysis," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Jan. 2000, pp. 241–150.

[13] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, Dec. 1989.

[14] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2007, pp. 23–34.

[15] S. Eyerman, L. Eeckhout, and J. E. Smith, "Studying compiler optimizations on superscalar processors through interval analysis," *Proceedings of the 2008 International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pp. 320–334, Jan. 2008.

[16] A. Aiken, "Moss: A system for detecting software plagiarism," `http://theory.stanford.edu/~aiken/moss/`.

[17] G. Malpohl, "JPlag: Detecting software plagiarism," `https://www.ipd.uni-karlsruhe.de/jplag/`.

[18] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *The Computer Journal*, vol. 19, no. 1, pp. 43–49, 1976.

[19] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984.

[20] C. Hsieh and M. Pedram, "Micro-processor power estimation using profile-driven program synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 11, pp. 1080–1089, Nov. 1998.

[21] C. Hughes and T. Li, "Accelerating multi-core processor design space evaluation using automatic multi-threaded workload synthesis," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Sept. 2008, pp. 163–172.

[22] L. Van Ertvelde and L. Eeckhout, "Dispersing proprietary applications as benchmarks through code mutation," in *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2008, pp. 201–210.

[23] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *Proceedings of the International Conference on Computer Design (ICCD)*, Oct. 1996, pp. 468–477.

[24] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge, "Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2005, pp. 78–88.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

[26] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, June 2003, pp. 84–95.

[27] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The University of Auckland, Tech. Rep. 148, July 1997.

Toward a More Accurate Understanding of the Limits of the TLS Execution Paradigm*

Nikolas Ioannou†, Jeremy Singer‡, Salman Khan†, Polychronis Xekalakis§ ∓, Paraskevas Yiapanis‡
Adam Pocock‡, Gavin Brown‡, Mikel Luján‡, Ian Watson‡, and Marcelo Cintra†

† School of Informatics ‡ School of Computer Science
University of Edinburgh University of Manchester

§ Intel Barcelona Research Center
Intel Labs Barcelona

ABSTRACT

Thread-Level Speculation (TLS) facilitates the extraction of parallel threads from sequential applications. Most prior work has focused on developing the compiler and architecture for this execution paradigm. Such studies often narrowly concentrated on a specific design point. On the other hand, other studies have attempted to assess how well TLS performs if some architectural/compiler constraint is relaxed. Unfortunately, such previous studies have failed to truly assess TLS performance potential, because they have been bound to some specific TLS architecture and have ignored one or another important TLS design choice, such as support for out-of-order task spawn or support for intermediate checkpointing.

In this paper we attempt to remedy some of the shortcomings of previous TLS limit studies. To this end a characterization approach is pursued that is, as much as possible, independent of specific architecture configurations. High-level TLS architectural support is explored in one common framework. In this way, a more accurate upper-bound on the performance potential of the TLS *execution paradigm* is obtained (as opposed to some particular architecture design point) and, moreover, relative performance gains can be related to specific high-level architectural support. Finally, in the spirit of performing a comprehensive study, applications from a variety of domains and programming styles are evaluated.

Experimental results suggest that TLS performance varies significantly depending on the features provided by the architecture. Additionally, the performance of these systems is not only hindered by *data dependences*, but also by *load imbalance* and limited *coverage*.

*This work was supported in part by EPSRC under grants EP/G000697/1 and EP/G000662/1 and by the EC under grant HiPEAC-2 IST-217068.

∓ Work performed while author was with the University of Edinburgh.

1 INTRODUCTION

As device scaling continues to track Moore's law and with the end of corresponding performance improvements in out-of-order processors, *multicore* systems have become the norm. Unfortunately, parallel programming is hard and error-prone, sequential programming is still prevalent, and compilers only auto-parallelize the most regular programs.

Thread-level speculation (TLS) [16, 21, 24, 36, 39] has long been proposed as a possible solution to this problem. In TLS systems the compiler/programmer is free to generate threads without having to consider all possible cross-thread data dependences. Parallel execution of threads then proceeds speculatively and the system guarantees the original sequential semantics of the program by transparently detecting data dependence violations, squashing offending threads, and returning the system to an earlier non-speculative correct state.

Research in the field has progressed in three major directions: architectural extensions (e.g., [8, 9, 10, 32, 34, 37, 38]), compiler support (e.g., [11, 12, 22, 33]), and program behavior characterization [18, 19, 25, 28, 29, 42]. Despite the significant forward progress in the field, the reception of TLS by industrial users is still lukewarm [1, 6].

While all the architectural extension proposals (e.g., [8, 9, 10, 32, 34, 37, 38]) have evaluated the benefits of possible extensions to the baseline TLS architecture, they fail to provide broader and more general insights. On the other hand, while works in program characterization [18, 19, 25, 28, 29, 42] have certainly provided some insight into the potential performance gains of TLS, they still fall short of providing an accurate evaluation of the potential gains of the TLS *execution paradigm*. The reasons for these shortcomings of previous work are threefold. (1) Proposals for new extensions naturally focused on the particular extension proposed and did not investigate how it interacts with other TLS features. Similarly, program behavior studies have focused on only a subset of TLS features and have not investigated how these interact. (2) Quantitative evaluations were often tied to a particular TLS architecture config-

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

uration and choice of parameters. (3) Benchmark choice was often limited to one particular domain or programming style.

In this paper we address these shortcomings and attempt to provide a better understanding of TLS performance potential. In particular, we provide a study that is, as much as possible, independent of any specific architecture configuration or choice of parameters. Instead, a variety of high-level architectural extensions is explored. In this way individual and combined features can be linked with their respective contribution to the potential performance improvement. Moreover, this study is based on a wide variety of benchmarks coming from the engineering, multimedia, and scientific domains and are written in C, Fortran, and Java.

To summarize, the contributions of this paper over previous TLS studies are as follows:

- We perform an in-depth *implementation-independent* study of TLS performance potential, which more accurately reflects the potential of the *execution paradigm*.
- We evaluate how individual *high-level TLS architectural features* contribute to overall performance gains, both in isolation and combination.
- We evaluate benchmarks from a *variety of application domains and programming styles*.

Experimental results suggest that TLS performance varies significantly depending on the features provided by the architecture. As expected, the benefit of certain features depends on the application domain and the programming style. The benefit also depends on the presence, or not, of other features. Finally, our results confirm that *data dependences* are not the only factor constraining performance. *Load imbalance* and limited *coverage* are also important factors in realizing the full performance potential of TLS.

The rest of this paper is organized as follows. Section 2 provides a brief description of TLS. Section 3 describes the methodology used in our study. Section 4 presents results. Finally, Section 5 discusses related work and Section 6 concludes the paper.

2 BACKGROUND

2.1 Basic TLS Execution

Under the *thread-level speculation* (also called *speculative parallelization* or *speculative multithreading*) approach, sequential sections of code are speculatively executed in parallel hoping not to violate any sequential semantics [16, 21, 24, 36, 39]. Sequential control flow imposes a total order on the threads. The terms *predecessor* and *successor* are used to relate threads in this total order. Stores in speculative threads generate unsafe *versions* of variables that are stored in a *speculative buffer*. As execution proceeds, the system tracks memory references to identify any cross-thread data dependence violation. Any value read from a predecessor thread is called an *exposed read*, and must be tracked since it may expose a read-after-write (RAW) dependence. If a dependence violation is found, the offending thread must be *squashed*, along with its successors. When the execution of a non-speculative thread completes it *commits* and the values it generated can be moved to safe storage. At this point its immediate successor acquires non-speculative status and is allowed to commit. When a speculative thread completes it must wait for all predecessors to commit before it can commit.

Speculative threads are usually extracted from either loop iterations or method continuations. The compiler marks these structures with a *spawn instruction*, so that the execution of such an instruction leads to a new speculative thread. The *parent* thread continues execution as normal, while the *child* thread is mapped to any available core. For loops, spawn points are placed at the beginning of the loop body, so that each iteration of the loop spawns the next iteration as a speculative thread. For method calls, spawn points are placed just before the method call, so that the non-speculative parent thread proceeds to the body of the method and a speculative child thread is created from the method's continuation.

2.2 Architectural Extensions

Below we outline the most important, previously proposed, architectural extensions to the basic TLS execution. In Section 4 we evaluate the potential performance benefits of all of these extensions quantitatively.

2.2.1 Multiversioned Cache

A *Speculative Versioning Cache* (SVC) [15] can hold state from multiple tasks by tagging each cache line with a version ID, which identifies the task to which the line belongs to. Multiversioned caches are beneficial in two ways: they avoid processor stalls when tasks are imbalanced, and enable lazy commit. If tasks have load imbalance, a processor may finish a task while it is still speculative. If the cache can only hold state for a single task, the processor has to stall until the task becomes safe. An alternative is to move the task state to some other buffer, but this leads to a more complicated design than a multiversioned cache. Lazy commit is an approach where, when a task commits, it does not eagerly merge its cache state with main memory. Instead, the task simply passes the commit token to its successor. Its state remains in the cache and is lazily merged with main memory later, usually as a result of cache line replacements.

2.2.2 Out-of-Order Spawn

In early TLS proposals, threads are formed in-order from iterations from a single loop level or the continuation of subroutine calls. This may significantly limit the amount of TLP that can

be exploited. Recently, architectural support for task creation from nested loops and nested methods has been proposed [34]. This optimization is called *out-of-order* (OoO) *spawning*, since the threads are spawned in a different order than their sequential semantics. Thread ordering for these systems is typically maintained via splitting timestamps [34].

2.2.3 Dynamic Dependence Synchronization

Dynamic dependence synchronization [27] is an important optimization for TLS systems, since it removes much of the overhead associated with recurring dependence violations. The idea is to dynamically detect which instructions violate the sequential semantics and force them to wait until the datum can be forwarded to them. Unfortunately, one cannot indiscriminately synchronize on all memory operations, since each time this is erroneously done, the consumer part of the dependence chain is unnecessarily stalled. We thus have to correctly identify the dependence chains. Most predictors studied so far in the literature are simple, table-based predictors indexed by either the Program Counter of the offending instructions [27] or their target memory addresses [9, 38].

2.2.4 Intermediate Checkpointing

Intermediate checkpointing schemes aim to reduce the misspeculation penalty by allowing partial rollback. They do so by trying to detect the violating loads and checkpoint the processor state *just before* these loads are executed. These schemes are thus guided from a dependence predictor [41], or some other heuristic (e.g., periodically, after a certain number of instructions have executed since the last checkpoint [10]). Once a positive prediction is obtained, the processor state is checkpointed and a subthread is created which has as its first instruction the predicted load. It is important to note that ignoring overheads and prefetching, perfect synchronization (Section 2.2.3) and perfectly placed checkpoints have the same effect.

2.2.5 Data Value Prediction

Normally in TLS systems values have to be forwarded from less speculative to more speculative threads. This can happen either by means of implicit forwarding (possibly following a squash and restart) or explicit synchronization (Section 2.2.3). There are, however, cases where due to value locality this is not necessary, since the values to be communicated can be easily predicted [9, 38]. In such cases squashes can also be avoided if the predicted value of the datum prematurely read is the same as the one that is subsequently stored, and as such the squash dictated by the TLS protocol is unnecessary.

2.2.6 Return Value Prediction

Another form of value prediction that is more specific to method-level speculation is return value prediction [7]. Under this optimization, the return value of a method is predicted. Although this optimization can be seen as a specialization of value prediction, it has been shown to be a fairly important performance optimization on its own.

2.3 Compilation Extensions

Next we outline the most important, previously proposed, extensions to the basic TLS compilation schemes. In Section 4 we evaluate a subset of these extensions.

2.3.1 Extracting Threads Beyond Program Structures

While most previous work on TLS has considered only threads extracted from high-level programming constructs, such as loops and methods, some works have considered other sources. For instance, the work in [17] considers all basic block boundaries as possible thread spawn points and uses a min-cut algorithm based on expected dependences to find the best selection of threads. Similarly, the work in [26] considers all control quasi-independent points (i.e., points more or less guaranteed to postdominate a certain program point) in the program as possible thread spawn targets. Such thread extraction approaches add flexibility to the traditional loop and method approaches, but also add significant complexity to the TLS compilation process. We leave a limit study of such extensions to future work.

2.3.2 Profiling Based Thread Selection

Given the unpredictability of TLS execution, in particular due to data dependence violations, the vast majority of compilation schemes for TLS have used profiling to select the most profitable threads [13, 22, 33]. As is the general case with profiling approaches to compilation, these are affected by run-time variation with different input sets, but have proven very effective in practice. In Section 4 we evaluate the effect of a simplified profiling based selection of loops from different nesting levels.

2.3.3 Thread Selection with Static Cost Analysis

Despite the high unpredictability of TLS execution, some attempts have been made toward static thread selection [12, 20, 40]. Most of these use simple heuristics, such as expected thread sizes, to select threads most likely to be profitable [20, 40], while others have used more complex cost functions [12]. In this work we do not explicitly consider the effects of such static analysis, but our limit study with perfect thread selection from different nesting levels somewhat subsumes part of the gains possible with such techniques.

2.3.4 Pre-Computation Slices

Some previous works have proposed the use of pre-computation slices for handling statically known data dependences [33]. The idea is to add code to be executed at the beginning of a thread to generate the live-in values and avoid data

Imperative (C/Fortran)			Object-Oriented (Java)	
SPECINT2006	**SPECFP2006**	**MediabenchII**	**SPECJVM98**	**DaCapo**
400.perlbench	416.gamess	mpeg4enc	_201_compress	antlr
401.bzip2	454.calculix	mpeg4dec	_202_jess	bloat
403.gcc	459.GemsFDTD	mpeg2dec	_205_raytrace	fop
429.mcf	465.tonto	jpg2000enc	_209_db	pmd
458.sjeng	470.lbm	jpg200dec	_213_javac	
462.libquantum	482.sphinx3	cjpeg	_222_mpegaudio	
		djpeg	_228_jack	

Table 1: Benchmarks.

dependence violations. We do not explicitly consider this compilation extension, but our limit study with perfect value prediction somewhat subsumes the gains possible with such techniques.

2.3.5 Compiler Inserted Dependence Synchronization

Some previous works have proposed the use of statically placed dependence synchronization to avoid data dependence violations [38, 43]. These techniques are not as flexible as dynamic approaches to synchronization (Section 2.2.3) but require simpler hardware. We do not explicitly consider this compilation extension, but our limit study with perfect synchronization completely subsumes the gains possible with such technique.

3 METHODOLOGY

In this section we present our simulation methodology for evaluating TLS and the proposed architectural features listed in the previous section. There are conceptually four steps in our methodology: (1) workload selection, (2) annotation of applications to produce (3) sequential traces, and (4) performance evaluation from the traces. The remainder of this section presents each step in detail.

3.1 Benchmarks

Having a representative workload is of paramount importance for any limit study, and as such we feel that this is one of the most important steps in our methodology (and a fundamental limitation in previous ones). An overview of the selected benchmarks can be seen in Table 1. We broadly categorize the chosen benchmarks in imperative applications (C and Fortran) and object-oriented ones (Java).

For the imperative applications, we use benchmarks from the SPEC CPU 2006 [2] benchmark suite running the train data set and from the Mediabench II [14] benchmark suite running the default data set. All benchmarks are compiled with optimization level `O2`.

Our object-oriented applications are Java benchmarks from the SPEC JVM 98 [3] and DaCapo [5] benchmark suites. We use data sets supplied along with the suites, specifically `s1` for SPEC JVM 98, and `small` for DaCapo.

3.2 Instrumentation

The tasks are selected from high-level program structures, consisting of loop iterations and method call continuations. We annotate such program structures at compile time. We chose two distinct compiler infrastructures to perform our experiments, one for the C and Fortran benchmarks and one for the Java ones.

For C and Fortran benchmarks we have used the *GNU Compiler Collection* (GCC) v4.3.3.We have created a compiler pass to annotate loop iterations and method call bodies. Register spilling is done at task boundaries so that all inter-task register dependences are communicated through memory. References to loop induction and reduction variables are also marked in the compiler and dependences carried by these references are not considered, since such dependences can be removed by compiler transformations [30]. The use of return values is also marked. Our annotation operates on the intermediate representation (tree SSA) of GCC at the end of the loop optimization passes, so loops, induction and reduction variables might not directly correspond to the original source code. We chose this option because we think it is more realistic to use optimized code instead of instrumenting at source level and interfering with compiler optimizations.

For Java benchmarks we have modified the Jikes RVM compiler v2.9.3 to instrument the object code. Again, we annotate loop iterations, method call bodies, and references to loop induction and reduction variables. Because Jikes RVM is an adaptive compilation system, we execute each Java benchmark for several warm-up iterations before instrumenting the code on a steady-state iteration, when hot methods have been compiled to a higher optimization level. Finally we eliminate all JVM runtime activity (garbage collection, etc) from the sequential trace files.

3.3 Trace Generation

Having the benchmarks annotated by the compiler, we next create the execution traces. The sequential benchmark traces were produced using Simics [23], a full-system functional simulator. The simulated processor is a single-issue in-order x86 core, where each instruction takes one cycle to execute. The simulator recognizes magic instruction sequences which execute call-backs from the simulated application to the simulator. We employ these to mark dynamic events in program execution, such as method call boundaries. We use the Simics memory profiling infrastructure to trace memory accesses. For C and Fortran benchmarks, we record all memory accesses. On the other hand, for Java benchmarks we only record (global) accesses into the heap, and stack-based accesses occurring directly in loop iterations. Note that in Java, a method's stack frame is guaranteed to be private to that method. Each event in the trace file has an associated sequential time-stamp, derived from the Simics cycle counter, which is adjusted to remove the overhead of each instrumentation code sequence.

3.4 Trace-Driven Simulation

The last step in our methodology is to feed the produced traces into our simulation infrastructure. In order to evaluate the performance impact of different high-level TLS architectural features we have crafted a trace-driven simulation tool. The tool parses the trace file, extracting threads along the way out of loop iterations and/or method call continuations, runs them in parallel with an infinite size *speculative buffer* and dynamically detects data dependences.

3.4.1 TLS Architecture Model

The architectural design options specific to TLS that our framework is able to accurately simulate cover what we believe are the most important TLS-related hardware optimizations. More specifically, using our tool we are able to simulate the effect of: multiversioned caches, out-of-order thread spawning, dynamic dependence synchronization, checkpointing, and data and return value prediction.

For dynamic dependence synchronization, data and return value prediction, the predictors we simulate are perfect. This choice was made because we only want to show an upper bound on performance relative to each feature, as opposed to evaluating different kinds of predictors. Note that, because we do not simulate prefetching effects, perfect synchronization and no overhead checkpointing have the same net effect on performance, such that the results presented for perfect synchronization in Section 4 actually refer to both.

3.4.2 Task Selection

The optimal partitioning of programs into non-speculative threads given the communication costs and the amount of computation associated with each method invocation, has been shown to be NP-Complete [35]. The partitioning of speculative tasks is further complicated because dependence and communication information is not complete. Thus, we rely on heuristics to perform task selection using both loop iterations and method call continuations. In the case of in-order loop-level speculation we consider two alternatives. The first is to evaluate innermost loops. The second is to choose the best loops to speculate on from three different dynamic depth levels, which somewhat emulates the result of an optimal profile or cost analysis driven compiler loop selection. In the case of out-of-order spawning, the dynamic thread spawning policy employed was one favoring the least speculative threads [34]. The spawning policy also takes into account the maximum thread size, and only spawns threads whose length is below a threshold. The threshold was chosen empirically based on the resulting speedup and is different for loop and method-level speculation. For loop-level speculation, we have used an upper bound of 500 instructions, effectively choosing smaller tasks. For method-level speculation we have used an upper bound of one million instructions, with smaller thresholds providing limited coverage.

4 EXPERIMENTAL RESULTS

4.1 Loop-Level Speculation

Loops have traditionally been the center of the research community's effort for parallelization and, hence, are the obvious choice for thread-level speculation as well. We first consider the simpler case of speculatively parallelizing the innermost loops only and then evaluate the potential of choosing between multiple dynamic loop depth levels.

4.1.1 In-Order Loop Speculation

Initially, from each loop nest we only choose the innermost loop to speculate upon. This is the most commonly studied case and is also the starting point of our evaluation. We speculatively parallelize *all* innermost loops in each of the benchmarks.

The results for this type of speculative parallelization are shown in Figure 1. It can be seen that there is significant potential from innermost loops for *lbm*, *libquantum* and *cjpeg*. That is because those benchmarks have high coverage parallel inner loops (99%, 94% and 95%, respectively). The lower scaling of *cjpeg* at 16 processors, however, can be attributed to low iteration counts for some of the innermost loops as well as one loop with 12% coverage that shows infrequent dependences. Detailed high coverage loop information for selected benchmarks is presented in Table 2.

Multiversioning provides minimal to no improvement over base TLS for this type of speculation. This comes as little surprise, since we did not expect innermost loops to be particularly load imbalanced (Section 4.4 provides further discussion on this).

Synchronizing around data dependences benefits *lbm* strongly. The reason is that even though the loop with 90% coverage is parallel, the remaining execution requires synchronization to obtain overlap, most significantly the two loops with the next higher coverage (4% and 2%) [1] require synchronization to be parallelized. Other benchmarks that benefit from synchronization are *mpeg2dec*, *djpeg* and, to a lesser degree, *gamess*. *mpeg2dec* has a 28% coverage loop which shows a speedup of 11 for 16 processors only if synchronized. Similarly, *djpeg* has a 26% coverage loop that shows no parallelism unless synchronized around data dependences. *gamess* has a 22% coverage loop that benefits significantly from synchronization.

Interestingly, value prediction does not seem to provide much benefit on top of synchronization. This suggests that for such fairly regular inner loops the producer and consumer in a data dependence occur very close in time such that synchronization incurs minimal overhead.

Unsurprisingly, Java applications do not show much potential from innermost loops. This is due to low coverage of in-

[1] Since 90% of the program is already parallel, these loops, which might appear to be unimportant, can now make a large contribution to speedup.

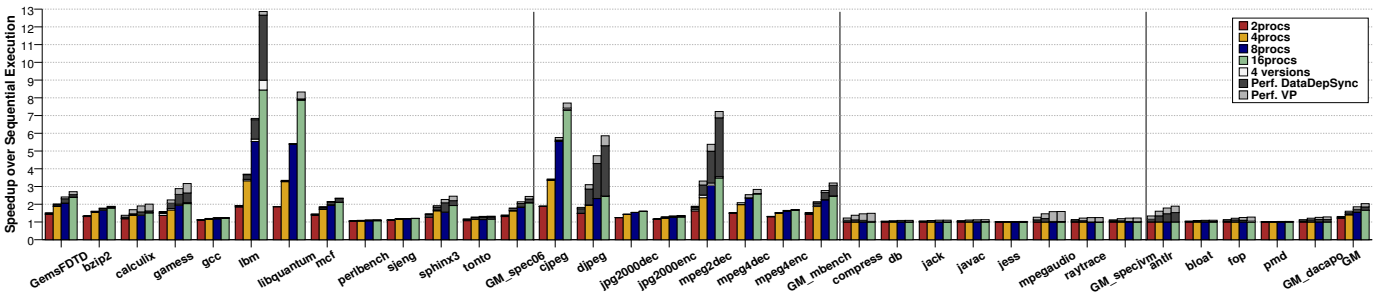


Figure 1: Speedup obtained from In-Order innermost Loop-Level Speculation. Each bar corresponds to a different number of processors. Stacked on top of the bars are the improvements in speedup obtained from 4 versions instead of 1, Perfect Data Dependence Synchronization and Perfect Value Prediction, respectively.

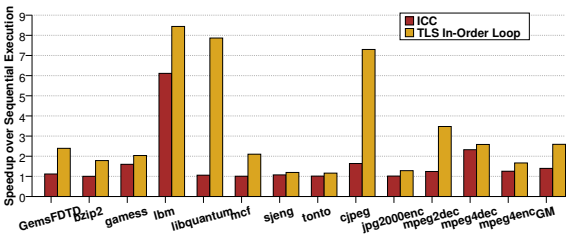


Figure 2: Comparison between parallel loops detected by ICC and TLS In-Order innermost Loop-Level speculation.

nermost loops for the benchmarks evaluated, ranging from 4% (*jess*) to 51% (*antlr*), with an average of 23%.

To verify that the speedup obtained through speculative parallelization is not simply through loops that are parallelizable through current compiler techniques, we compare with Intel's ICC compiler in Figure 2. For a fair comparison, we compute the speedup obtained by the compiler by simulating the same way as earlier, but only parallelizing loops detected as parallel by the compiler. The only benchmarks that the compiler is able to automatically parallelize to any significant degree are *lbm* and *mpeg4dec*.

Beyond innermost loops we also evaluate higher levels of nesting. We do this by picking the best performing level [2] of the first three dynamic loop depths. Figure 3 depicts the results for the best performing loops in a nesting level. *libquantum* continues to get most of its speedup from the high coverage inner loop. However, considering higher nesting levels adds a number of other lower coverage loops which takes the total coverage from 94% to 99%. Similarly, for *mcf* the coverage of the loops contributing is increased from 64% to 94%. In the case of *djpeg*, adding higher level loops increases the coverage and the speedup in the base case. However, these higher level loops suffer from low iteration counts which limit the scalability; this explains why there is no longer any improvement from synchronization or perfect value prediction. For *jpg2000enc* we see a dramatic increase in coverage from 42% to 98% from considering outer loops; these new loops, however, need synchronization and value prediction for realizing their potential. The Java benchmarks continue to be mostly coverage limited and, in some cases (e.g., *mpegaudio*) have high coverage loops with limited speedup due to load imbalance.

[2] When picking the best level we don't take into account synchronization or value prediction.

4.1.2 Out-of-Order Loop Speculation

In an effort to better exploit loop level parallelism, we also evaluate simultaneously speculating on multiple levels of the same loop nest. This entails spawning out-of-order tasks. The task selection performed is the one presented in Section 3.4. Picking loops to speculate on in this way, we see mixed results, shown in Figure 4. Because we favor safer threads, in loop-level speculation this translates to choosing inner loops most of the time. The only case where this differs is when the inner loops suffer from low iteration count. If the iteration count is low enough, and given enough cores, then the outer loop is speculatively parallelized. In some cases this is beneficial, like *sjeng*, *cjpeg*, *mpeg2dec*, and the *jpg2000* benchmarks. In other cases, if the thread size of the outer loop is significantly different from the inner loop this may result in load imbalance. We observe a slowdown due to this effect compared to choosing the best single level per nest for *libquantum*, *mcf*, and *sphinx3*. The increase in the improvement observed due to multiversioning is also a repercussion of the load imbalance imposed, in some cases, by the task selection policy.

4.2 Method-Level Speculation

4.2.1 In-Order Method Speculation

We evaluate speculatively overlapping method continuations with the methods. We assume perfect return value prediction for methods. As seen in Figure 5, in-order method level speculation is constrained by real dependences that lead to serialization in the presence of synchronization. Data value predic-

Program	Loop Location	Coverage	Depth	Avg Iter. Count	Avg Size	TLS Speedup	Perf. Data. Dep. Speedup	Perf. VP. Speedup
lbm	lbm.c:187	90.9	1	1300K	262	12.7	12.7	12.7
	lbm.c:460	4.04	1	1300K	147	1.81	14.4	14.8
	lbm.c:105	2.10	1	100	257	1.36	7.82	14.1
libquantum	gates.c:96	56.3	1	2048	16	15.4	15.4	15.4
	gates.c:65	15.2	1	2048	36	16.0	16.0	16.0
	gates.c:174	11.7	1	2048	17	16.0	16.0	16.0
	gates.c:NA	7.6	2	2048	358	16.0	16.0	16.0
gamess	rhfuhf.fppized.f:410	22.7	1	13K	186	1.91	6.88	15.9
mcf	pbeampp.c:167	40.7	1	300	22	14.3	14.4	14.4
	implicit.c:285	17.0	2	3K	56	2.49	5.50	10.6
	pbeampp.c:81	16.3	2	7	21	3.35	3.43	3.44
cjpeg	jccolor.c:149	33.4	1	704	50	16.0	16.0	16.0
	jcdctmgr.c:185	18.1	1	64	18	14.7	14.7	14.7
	jccoefct.c:144	44.8	3	1.33	4K	1.33	1.33	1.33
	jchuff.c:477	12.7	1	64	23	2.42	8.03	11.7
	jfdctint.c:220	10.9	1	8	88	7.92	7.92	7.92
	jfdctint.c:155	10.5	1	8	84	7.92	7.92	7.92
djpeg	jdcolor.c:145	38.4	1	704	95	16.0	16.0	16.0
	jdsample.c:379	26.2	1	350	108	1.03	7.57	15.9
	jidctint.c:278	15.2	1	8	80	7.91	7.91	7.91
	jdsample.c:378	26.2	2	2	9K	2.00	2.00	2.00
	jdcoefct.c:193	27.6	3	1.33	1.7K	1.33	1.33	1.33
jpg2000enc	jpc_t1ecn.c:871	19.1	2	57	1.8	1.24	2.20	9.57
	jpc_t1enc.c:472	15.6	2	56	2.7K	1.48	1.57	10.3
	jpc_cs.c:941	14.7	3	704	6.5K	1.23	1.53	16.0
	jpc_qmfb.c:595	8.65	2	248	78K	1.37	4.19	16.0
	jpc_dec.c:1091	7.54	2	794	187	1.61	10.6	15.7
mpeg2dec	store.c:259	28.9	1	704	360	1.79	11.4	15.9

Table 2: High coverage loop information for selected benchmarks. The speedup numbers presented are for 16 processors and the loops for each benchmark are ordered based on their Perfect Value Prediction speedup contribution.

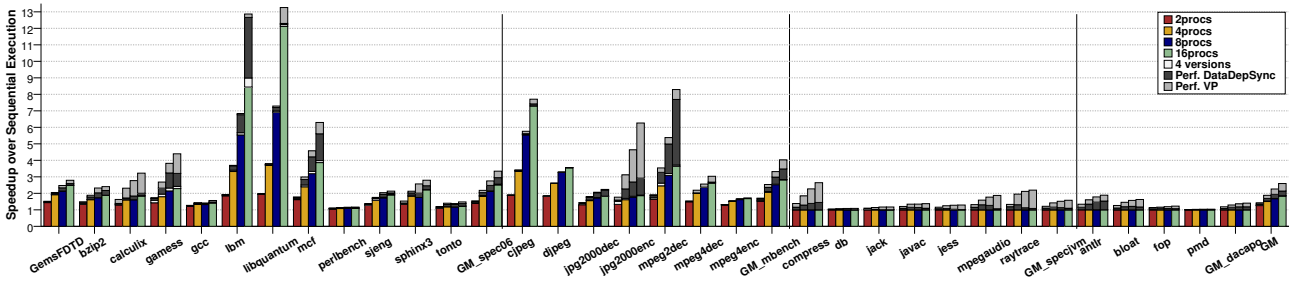


Figure 3: Speedup obtained from choosing the best loop for each loop nest out of three loop depth levels.

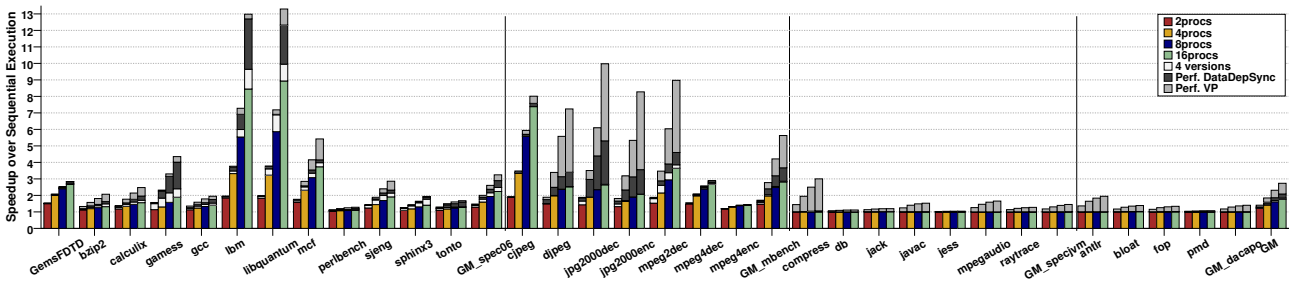


Figure 4: Speedup obtained from Out-of-Order Loop-Level Speculation.

tion has to be employed if significant performance is to be obtained, with the *specjvm98* benchmarks showing the most potential. *mpegaudio* is an exception to this, and shows reasonable speedup with base TLS that can be significantly improved with multiversioning.

4.2.2 Out-of-Order Method Speculation

Figure 6 shows out-of-order method level speculation. This means that nested method calls can be speculated upon as well. This leads to an improvement in speedup with base TLS for a number of benchmarks compared to in-order method-level speculation. Multiversioning is essential for realizing this. However, we no longer see significant improvement from value prediction for most of the benchmarks, suggesting that our OoO task selection leads to increased load imbalance and/or lower coverage. Looking ahead to Figure 9 in Section 4.4, we see that OoO method-level speculation exhibits lower coverage than in-order method-level speculation (e.g., *gcc*, *libquantum*, *jess*, *pmd*, etc). The two exceptions to the decrease in value prediction potential are *djpeg* and *cjpeg*, which exhibit both better base TLS speedup *and* increased potential from value prediction. Moreover, synchronization is also very beneficial in those two cases.

The difference in behavior between in-order and OoO method level speculation is to be attributed to the task selection that each of them entails. In the in-order case, when a method is chosen to be speculatively parallelized, only its continuation (being the most speculative task) is allowed to further spawn more speculative tasks and this is recursively repeated until we fill the processors contexts, assuming enough tasks. On the other hand, OoO coupled with our greedy task selection policy that favors safer threads, will mostly spawn speculative tasks *within* the chosen method(s), assuming enough method calls within the speculated method(s).

4.3 Mixed Speculation

Speculating at both loop iterations and method call continuations is the next form of speculation we consider. The spawning policy we employ in this case is a run-time out-of-order one similar to the one used in loops and methods separately. The difference now is that we can spawn threads from both types of program structures at the same time. We retain the static thread size heuristic that showed most potential, i.e., the one used for loops for SPEC and Mediabench and the one used for methods for the Java benchmarks.

As can be derived from Figures 7 and 8, applying a joint speculation mode can combine the benefits of loop and method level speculation. For most of the benchmarks we simply retain the benefit obtained from either loop-level or method-level speculation. In some cases we see combined performance improvements (*gcc*, *mpeg2dec*, and *mpeg4*). However, we see negative effects from joint speculation for the *mpegaudio* and, to a lesser extent, *jpg2000* benchmarks. If we compare the results obtained with in-order against OoO for mixed speculation we can see that OoO spawning, with the greedy task selection policy employed, does not provide notable benefits over in-order for most of the benchmarks.

4.4 Limits on Performance

The overriding pattern to be seen in the results shown in the previous sections is that performance is not constrained primarily through *data dependences*. Although some of the benchmarks show significant improvement through perfect value prediction, many of them do not. This fact points toward the conclusion that we are also limited by *low coverage* (i.e., large portions of the code are not chosen for speculative execution either because of the granularity of the tasks or because of a task selection heuristic) and *load imbalance* (i.e., co-scheduled speculative tasks have diverse sizes).

As a first step to understanding the performance limitations that handicap speculative execution we try to quantify the effect of *load imbalance*. We compute *load imbalance* as the relative difference between the theoretical maximum speedup obtained using Amdahl's law [4] and the speedup obtained with perfect data value prediction. The intuition behind this is that in the presence of perfect value prediction the only limiting factor to achieving the theoretical speedup is *load imbalance*. The coverage used in the calculation of Amdahl's law is the combined coverage of speculated loops for in-order loop speculation and

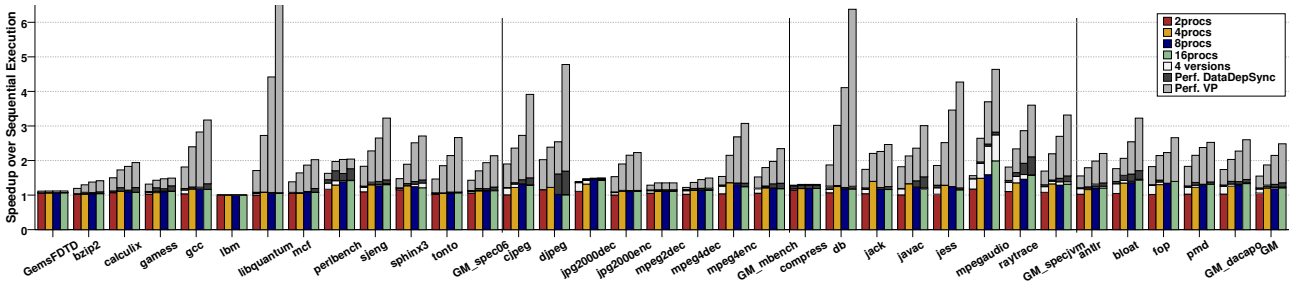


Figure 5: Speedup obtained from In-Order Method-Level Speculation.

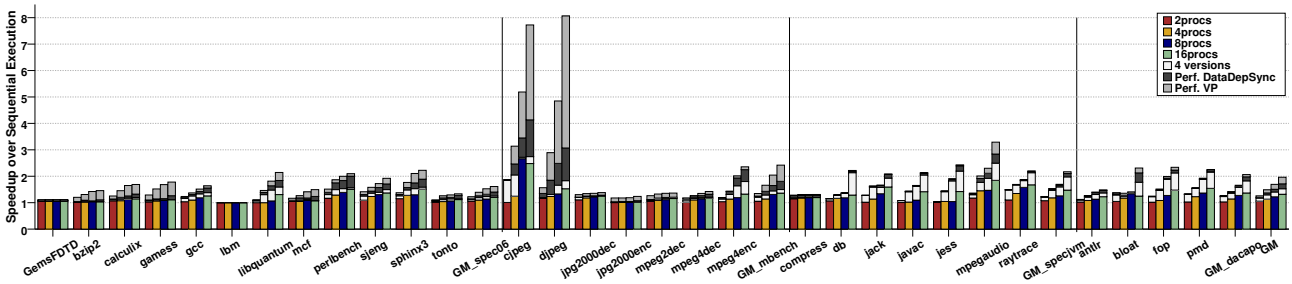


Figure 6: Speedup obtained from Out-Of-Order Method-Level Speculation.

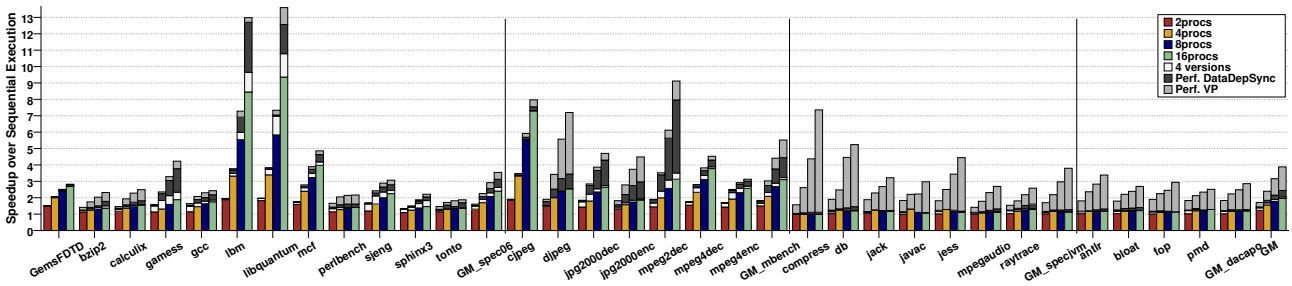


Figure 7: Speedup obtained from In-Order Mixed Speculation.

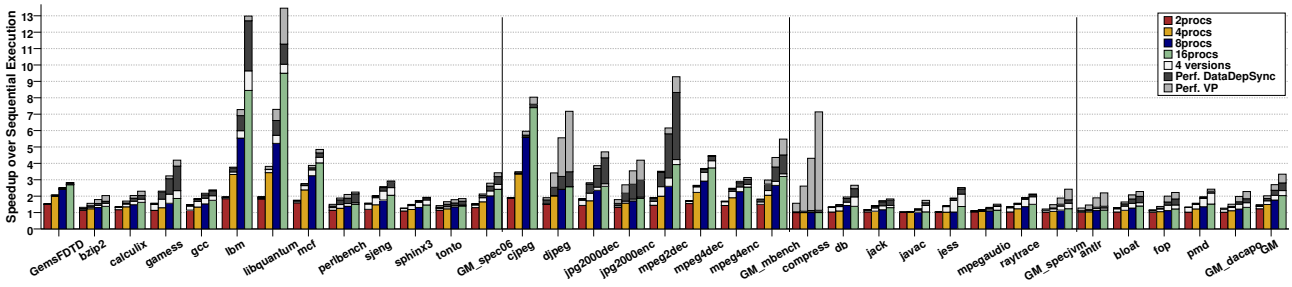


Figure 8: Speedup obtained from Out-Of-Order Mixed Speculation.

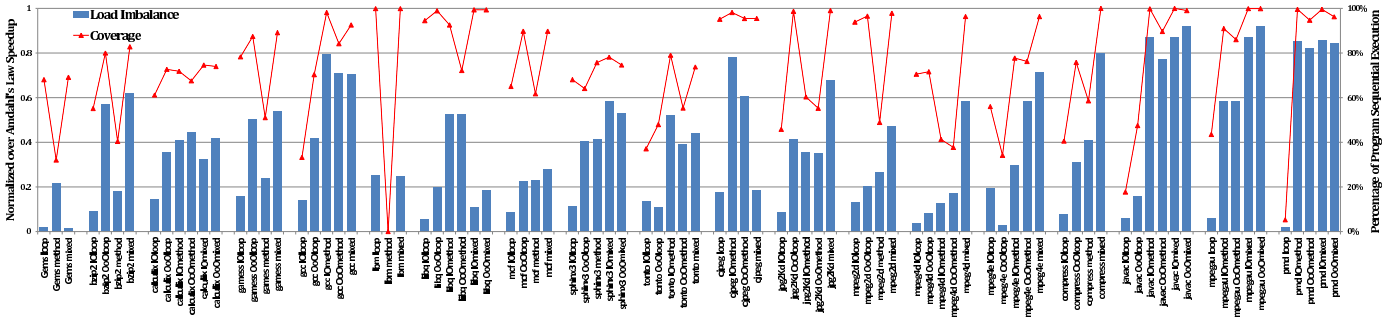


Figure 9: Relative Load Imbalance (bars) and Coverage (line), for 16 processors. For benchmarks showing nearly identical behavior we show only a representative one. Identical OoO and In-Order results are also merged.

the combined coverage of the outermost speculated regions in all other speculation types [3].

In Figure 9 *load imbalance* and *coverage* are plotted, with the different speculation types clustered together for representative benchmarks [4]. Near-ideal TLS speedup is indicated with a high coverage score (red triangle) and a low imbalance (blue bar). Examining the data points of this graph for each of the speculation types against their respective speedup results in the previous sections can provide insight into performance limitations. Let us take *jpg2000dec*, for example. The low potential from innermost loops can be completely attributed to low coverage (*jpg2Kd IOloop* point in Figure 9). The significant increase in potential observed going from innermost loop speculation to OoO loop speculation is mostly because the coverage goes from 45% to nearly 100% (*OoOloop*). Method speculation does not provide any benefits for *jpg2000dec* and this is reflected in the relatively low coverage with increased load imbalance (*IOmethod* and *OoOmethod*). Mixed speculation, although showing near 100% coverage on speculated regions, has its performance potential hindered by the inclusion of imbalanced methods (*mixed*).

Some benchmarks have near 100% coverage, but do not achieve corresponding TLS speedups, even with perfect value prediction. This problem is caused by extreme *load imbalance*. As mentioned in Section 2.2.1, multiversioned caches can potentially alleviate the problem of load imbalance. To verify this, we simulated with 16 processors and 200 versions. Representative cases, featuring one benchmark from each benchmark suite, are shown in Figure 10. In some cases, multiversioning is successful in minimizing *load imbalance* and thus unlocking value prediction and, to a lesser extent, synchronization potential (*cjpeg IOmethod*, *cjpeg OoOmethod*, and *pmd IOmixed*). In these cases, however, we see no improvements in the base-

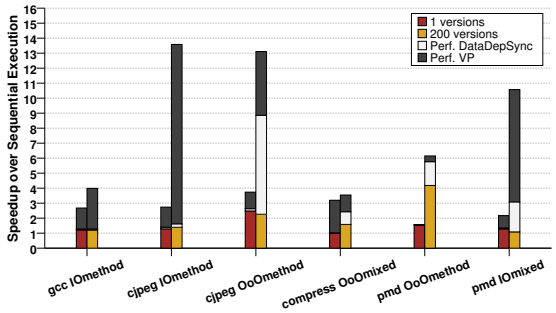


Figure 10: Performance impact of multiversioning for selected benchmarks and speculation types showing high coverage and high load imbalance, on 16 processors.

line performance. This is due to data dependences between the speculative tasks. In the case of *pmd OoOmethod* multiversioning realizes performance improvement for base TLS, indicating the absence of data dependences. Finally, multiversioning is unable to provide any significant performance benefits for *gcc IOmethod* and *compress OoOmixed*. This indicates toward *load imbalance* that is inherent in the program structures chosen for task extraction. Task selection is key for improving performance in such cases.

5 RELATED WORK

One of the first studies to attempt to establish performance upper-bounds of TLS in an ideal setting was [28]. In that study no compiler support particular to TLS was considered, and as such compiler optimizations, like induction variable elimination, were not considered. The work in [25], evaluated different design parameters such as speculation type (method and loop) and the order of spawning threads. A significant limitation of that work is that the spawning policies evaluated are considered independently and not concurrently, while they do not consider outer loops (a significant source of speedup). The work in [42] investigates the limits of method-level TLS and tries to establish the degree at which it varies between the imperative and

[3] In doing so we assume that the speculated regions can be partitioned into at least n parallel tasks, where n is the number of processors.

[4] The following subsets are formed based on load imbalance and coverage behavior: {*pmd*, *bloat*, *fop*, *db*, *jess*, *ray*}, {*jpg2Kd*, *jpg2Ke*}, {*cjpeg*, *djpeg*}, {*gcc*, *perlbench*, *sjeng*}, {*javac*, *jack*}, {*mpegaudio*, *antlr*}, {*calculix*, *sphinx3*}

the object-oriented programming models. Unfortunately, loop-level parallelism is omitted from that study and thus much of the potential of TLS is not exposed.

More recently, the work in [19] investigated the limits of performance improvements that may come exclusively from TLS and that could not be achieved through compiler auto-parallelization. In a follow-up study [18] the authors also investigated the effects of various threading overheads and misspeculation penalties. However, both studies assumed capabilities that are well beyond those of current auto-parallelizing compilers. At the same time, by not considering for speculative parallelization outer loops and not allowing out-of-order spawning of threads, they understated the potential of TLS systems. The work in [29] investigated again the potential of TLS, attempting to parallelize *all* loops that can benefit from TLS. Albeit closer to a better evaluation of TLS systems, that study only looked at applications from the SPEC 2000 and SPEC 2006 benchmark suites and, thus, does not have a fully representative mix of workloads. A different approach on the limits of TLS has been taken in [31]. In that study the authors manually parallelize several sequential applications and elaborate on the code transformations needed to achieve different levels of TLS performance. Although the insight provided into the TLS performance with the assumption of high-level programmer intervention is interesting, the authors do not study TLS-specific architecture optimizations.

The work in [42] is the only one that we are aware of that does experiment with dynamic applications. However, the benchmarks were compiled statically instead of using a Java runtime environment.

Compared to previous work, we perform an evaluation of TLS on a broad and diverse range of workloads. Also complementary to the related work is our investigation of *load imbalance* and *coverage* within the context of TLS as well as an evaluation of multiversioned caches as a means of improving load imbalance.

6 CONCLUSIONS

As multicore systems become common, the burden of extracting performance has shifted from the hardware toward the compiler/programmer side. Unfortunately, parallel programming is still hard and error prone and, perhaps more importantly, there is still a large sequential legacy code base. TLS systems offer a compelling alternative, in that they relieve the programmer from this burden by speculatively parallelizing sequential applications and dynamically checking whether the parallelization is correct.

In this paper we address many of the limitations of prior work and show that there is a lot to be gained from synchronization and value prediction. We also show that *load imbalance* and limited *coverage* are major factors in realizing potential along with *data dependences*. Task selection is, therefore, extremely important. We perform a limited task selection evaluation with perfect thread selection from nesting levels for loop level speculation, and employ a greedy task selection policy for OoO spawning but a lot remains to be evaluated and further exploration of task selection issues is left for future work.

We evaluate how individual high-level TLS architectural features contribute to overall performance gains, both in isolation and combination. Our results indicate that OoO spawning, with the greedy task selection policy employed, does not provide much benefit over in-order for most of the benchmarks.

REFERENCES

[1] Azul Systems. Vega 3 Processor. http://www.azulsystems.com.

[2] SPEC CPU2006. http://www.spec.org/cpu2006.

[3] SPEC JVM98. http://www.spec.org/jvm98.

[4] G. M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". *Spring Joint Computer Conf.*, pages 483–485, Apr 1967.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". *Intl. Conf on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, October 2006.

[6] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. "Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor". *Intl. Symp. on Computer Architecture (ISCA)*, pages 484–495, June 2009.

[7] M. Chen and K. Olukotun. "Exploiting Method-Level Parallelism in Single-Threaded Java Programs". *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 176–184, Oct 1998.

[8] M. Cintra, J. Martínez, and J. Torrellas. "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors". *Intl. Symp. on Computer Architecture (ISCA)*, pages 13–24, June 2000.

[9] M. Cintra and J. Torrellas. "Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors". *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 43–54, February 2002.

[10] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. "Tolerating Dependences Between Large Speculative Threads Via Sub-Threads". *Intl. Symp. on Computer Architecture (ISCA)*, pages 216–226, June 2006.

[11] X. Dai, A. Zhai, W.-C. Hsu, and P.-C. Yew. "A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion". *Intl. Symp. on Code Generation and Optimization (CGO)*, pages 280–290, March 2005.

[12] J. Dou and M. Cintra. "Compiler Estimation of Load Imbalance Overhead in Speculative Parallelization". *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 203–214, September 2004.

[13] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. "A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs". *Conf. on Programming Language Design and Implementation (PLDI)*, pages 71–81, June 2004.

[14] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. "MediaBench II Video: Expediting the Next Generation of Video Systems Research". *Microprocessors & Microsystems*, volume 33, pages 301–318, June 2009.

[15] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. "Speculative Versioning Cache". *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 195–205, February 1998.

[16] L. Hammond, M. Willey, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor". *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 58–69, October 1998.

[17] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. "Min-Cut Program Decomposition for Thread-Level Speculation". *Conf. on Programming Language Design and Implementation (PLDI)*, pages 59–70, June 2004.

[18] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. "Tight Analysis of the Performance Potential of Thread Speculation Using SPEC CPU 2006". *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 215–225, March 2007.

[19] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. "On the Performance Potential of Different Types of Speculative Thread-Level Parallelism". *Intl. Conf. on Supercomputing (ICS)*, pages 24–35, June 2006.

[20] S. W. Kim and R. Eigenmann. "The Structure of a Compiler for Explicit and Implicit Parallelism". *Intl. Wksp. on Languages and Compilers for Parallel Computing (LCPC)*, August 2001.

[21] V. Krishnan and J. Torrellas. "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor". *Intl. Conf. on Supercomputing (ICS)*, pages 85–92, July 1998.

[22] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. "POSH: a TLS Compiler that Exploits Program Structure". *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 158–167, March 2006.

[23] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. "Simics: A Full System Simulation Platform". *IEEE Computer*, volume 35, pages 50–58, February 2002.

[24] P. Marcuello and A. González. "Clustered Speculative Multithreaded Processors". *Intl Conf. on Supercomputing (ICS)*, pages 77–84, June 1999.

[25] P. Marcuello and A. González. "A Quantitative Assessment of Thread-Level Speculation Techniques". *Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 595–601, May 2000.

[26] P. Marcuello and A. González. "Thread-Spawning Schemes for Speculative Multithreading". *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 55–64, February 2002.

[27] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. "Dynamic Speculation and Synchronization of Data Dependence". *Intl. Symp. on Computer Architecture (ISCA)*, pages 181–193, June 1997.

[28] J. Oplinger, D. Heine, and M. Lam. "In Search of Speculative Thread-Level Parallelism". *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 303–313, October 1999.

[29] V. Packirisamy, A. Zhai, W. Hsu, P. Yew, and T. Ngai. "Exploring Speculative Parallelism in SPEC2006". *Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 77–88, April 2009.

[30] S. Pop, A. Cohen, and G. Silber. "Induction Variable Analysis with Delayed Abstractions". *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 218–232. October 2005.

[31] M. K. Prabhu and K. Olukotun. "Exposing Speculative Thread Parallelism in SPEC2000". *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 142–152, June 2005.

[32] M. Prvulovic, M. Garzarán, L. Rauchwerger, and J. Torrellas. "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization". *Intl Symp. on Computer Architecture (ISCA)*, pages 204–215, June 2001.

[33] C. G. Quinones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. "Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices". *Conf. on Programming Language Design and Implementation (PLDI)*, pages 269–279, June 2005.

[34] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. "Tasking With Out-Of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation". *Intl. Conf. on Supercomputing (ICS)*, pages 179–188, June 2005.

[35] V. Sarkar and J. Hennessy. "Partitioning Parallel Programs for Macro-Dataflow". *Conf. on LISP and Functional Programming (LFP)*, pages 202–211, 1986.

[36] G. Sohi, S. Breach, and T. Vijaykumar. "Multiscalar Processors". *Intl. Symp. on Computer Architecture (ISCA)*, pages 414–425, June 1995.

[37] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "A Scalable Approach to Thread-Level Speculation". *Intl. Symp. on Computer Architecture (ISCA)*, pages 1–12, June 2000.

[38] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "Improving Value Communication for Thread-Level Speculation". *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 65–75, February 2002.

[39] J. G. Steffan and T. C. Mowry. "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization". *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 2–13, January 1998.

[40] T. N. Vijaykumar and G. S. Sohi. "Task Selection for a Multiscalar Processor". *Intl. Symp. on Microarchitecture (MICRO)*, pages 81–92, December 1998.

[41] M. Waliullah and P. Stenstrom. "Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems". *Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–11, April 2008.

[42] F. Warg and P. Stenstrom. "Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms". *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 221–230, September 2001.

[43] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. "Compiler Optimization of Scalar Value Communication Between Speculative Threads". *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 171–183, October 2002.

A Limit Study of JavaScript Parallelism

Emily Fortuna Owen Anderson Luis Ceze Susan Eggers

Computer Science and Engineering, University of Washington
{fortuna, owen, luisceze, eggers}@cs.washington.edu
http://sampa.cs.washington.edu

***Abstract*—JavaScript is ubiquitous on the web. At the same time, the language's dynamic behavior makes optimizations challenging, leading to poor performance. In this paper we conduct a limit study on the potential parallelism of JavaScript applications, including popular web pages and standard JavaScript benchmarks. We examine dependency types and looping behavior to better understand the potential for JavaScript parallelization. Our results show that the potential speedup is very encouraging—averaging 8.9x and as high as 45.5x. Parallelizing functions themselves, rather than just loop bodies proves to be more fruitful in increasing JavaScript execution speed. The results also indicate in our JavaScript engine, most of the dependencies manifest via virtual registers rather than hash table lookups.**

I. Introduction

In an increasingly online world, JavaScript is around every corner. It's on your search engine and your new word processing software [1]. It's on your smart phone browsing the web and possibly running the applications on your phone [2]. One company has estimated that 99.6% of sites online today use JavaScript [3]. Dubbed the "Assembly Language of the Internet," JavaScript is used not only for writing applications, but also as the target language for web applications written in other languages, such as Java [4], [5], [6].

At the same time, the very features that have made this dynamically-typed language so flexible and popular make it challenging for compiler writers to efficiently optimize its execution. A number of websites offer tips and services to both manually and automatically optimize JavaScript code for download time and execution performance [7], [8], [9]. Further, many web-based companies such as Google, go as far as optimizing versions of their websites specifically for mobile devices.

These JavaScript optimizations are of particular importance for online mobile devices, where power is rapidly becoming the limiting factor in the performance equation. In order to reduce power consumption, mobile devices are following in the footsteps of their desktop counterparts by replacing a single, higher-power processor with multiple lower-power cores [10]. At the same time, advances in screen technology are reducing power consumption in the display [11], leaving actual computation as a larger and larger proportion of total power usage for mobile devices.

There have been a number of efforts to increase the speed of JavaScript and browsers in general [12], [13], [14]. More recently, researchers have sought to characterize the general behavior of JavaScript from the language perspective in order to write better interpreters and Just-In-Time (JIT) compilers [15], [16], [17]. Several of these studies compared the behavior between JavaScript on the web to standard JavaScript benchmarks such as SunSpider and V8. Researchers unanimously concluded that the behavior of JavaScript on the web differs significantly from that of standard JavaScript benchmarks.

This paper addresses the slow computation issue by exploring the potential of parallelizing JavaScript applications. In addition to the obvious program speedups, parallelization can lead to improvements in power usage, battery life in mobile devices, and web page responsiveness. Moreover, improved performance will allow future web developers to consider JavaScript as a platform for more compute-intensive applications.

Contributions and Summary of Findings

We present the first limit study to our knowledge of parallelism within serial JavaScript applications. We analyze a variety of applications, including several from Alexa's top 500 websites [18], compute-intensive JavaScript programs, such as a fluid dynamics simulator, and the V8 benchmarks [19]. In contrast to prior JavaScript behavior studies [15], [16], [17], we take a lower-level, data-dependence-driven approach, with an eye toward potential parallelism.

As a limit study of parallelism, we focused on two fundamental limitations of parallelism: data dependences and most control dependences (via task formation; see Section III-A). We impose JavaScript-specific restrictions, such as the event-based nature of execution. We believe this model captures all the first order effects on parallelism.

Our results show that JavaScript exhibits great potential for parallelization—speedups over sequential execution are 8.9x on average and as high as 45.5x. Unlike high-performance computing applications and other successfully parallelized programs, however, JavaScript applications currently do not have significant levels of loop-level parallelism. Instead, better parallelization opportunities arise between and within functions. As a result, current JavaScript applications will require different parallelization strategies.

The vast majority of data dependencies manifest themselves in the form of virtual registers, rather than hash-table lookups, which is promising for employing static analyses to detect parallelism. Finally, our results demonstrate that parallel behavior in JavaScript benchmarks differs from that of real websites, although not to the extent found for other JavaScript behaviors measured in [15], [16].

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

The remainder of this paper explains how we reached these conclusions. Section II provides some background information on the JavaScript language. Section III describes the model we used for calculating JavaScript parallelism, special considerations in measuring parallelism in JavaScript, and how we performed our measurements. Next, Section IV details our findings with respect to JavaScript behavior and potential speedup. Section V describes related work and its relationship to our study, and Section VI concludes.

II. Background

JavaScript is a dynamically typed, prototype-based, object-oriented language. It can be both interpreted and JIT'ed by JavaScript engines embedded in browsers, such as SpiderMonkey in Firefox [20], SquirrelFish Extreme[1] in Safari [21], and V8 in Chrome [22]. The current standard for JavaScript does not directly support concurrency.

Conceptually, JavaScript objects and their properties are implemented as hash tables. Getting, setting, or adding a property to an object requires the interpreter or JIT to either look up or modify that particular object's hash table.

The majority of production interpreters[2] convert JavaScript syntax into an intermediate representation of JavaScript opcodes and use one of two main methods to manipulate the opcodes' operands and intermediate values: stacks or registers [23]. Stack-based interpreters contain a data stack (separate from the call stack) for passing operands between opcodes as they are executed. Register-based interpreters contain an array of "register slots" (that we call "virtual registers," which are higher-level than actual machine registers) that are allocated as needed to pass around operands and intermediate data. Whether an interpreter is stack-based or register-based does not affect the way that data dependencies manifest; these two methods are simply used for storing the operands for each opcode. Interpreters such as SpiderMonkey are stack-based, whereas SquirrelFish Extreme is register-based. Some research suggests that register-based interpreters can slightly outperform stack-based interpreters [24].

Memory in a JavaScript program consists of heap-allocated objects created by the program, as well as a pre-existent Document Object Model produced by the browser that represents the web page and the current execution environment. All of these are distinct from local variables and temporaries. In this study we examine a register-based interpreter, which therefore exposes dependencies via virtual registers for local variables and temporaries, and via hash table lookups when accessing an object's hash table.[3]

[1]Also known by its marketing name, Nitro.

[2]The notable exception is V8, which compiles JavaScript directly to assembly code.

[3]In this study, register slots that directly reference objects for hash table lookups we classify as hash table lookups, distinct from local variables and temporaries stored in the register slots that we classify as being stored in "virtual registers."

A Day in the Life of a JavaScript Program

For the most part, JavaScript is used on the web in a highly interactive, event-based manner. A JavaScript event is any action that is detectable by the JavaScript program [25], whether initiated by a user or automatically. For example, when a user loads a web page, an `onLoad` event occurs. This can trigger the execution of a programmer-defined function or a series of functions, such as setting or checking browser cookies. After these functions have completed, the browser waits for the next event before executing more JavaScript code.

Next, in our example a user might click on an HTML `p` element, causing an `onClick` event to fire. If the programmer intends additional text to appear after clicking on the `p` element, the JavaScript function must modify the Document Object Model (DOM), which contains all of the HTML elements in the page. The JavaScript function looks up the particular object representing the `p` element in the DOM and modifies the object's text via its hash table, causing the browser to display the modified page without requiring an entire page reload. Any local variables needed to complete this text insertion are stored in virtual registers.

III. Methodology

Because of JavaScript's event-based execution model described above, we must make special considerations to quantify the amount of potential speedup in JavaScript web applications. Below we explain some terminology for talking about JavaScript parallelism, how we set up our model (Section III-A), and how we made our measurements from the dynamic execution traces (Section III-B).

A. Event-driven Execution Model Considerations

For our study, we parallelize each event individually, as described in Section II by dividing them into *tasks* that can be potentially executed concurrently. We define a *task* as any consecutive stream of executed opcodes in an event, delimited either by a function call, a function return, an outermost[4] loop entrance, an outermost loop back-edge, or an outermost loop exit. A task consists of at least one JavaScript opcode and has no maximum size. Tasks are the finest granularity at which we consider parallelism; we do not parallelize instructions within tasks. Since the tasks are essentially control independent, we take into account control dependences indirectly.

Figure 1 illustrates a high-level representation of the tasks created from the execution of a simple JavaScript function. When executing function `f`, JavaScript code is converted into a dynamic trace of opcodes. Our offline analysis divides these opcodes into eight tasks, appearing in boxes on the right of Figure 1.

We define the *critical path* as the longest sequence of memory and/or virtual register dependencies between tasks that occurs *within a given event*. The critical path is the limiting factor that determines the overall execution time for an event; all other data-dependent sequences can run in parallel

[4]Not nested inside any surrounding loop.

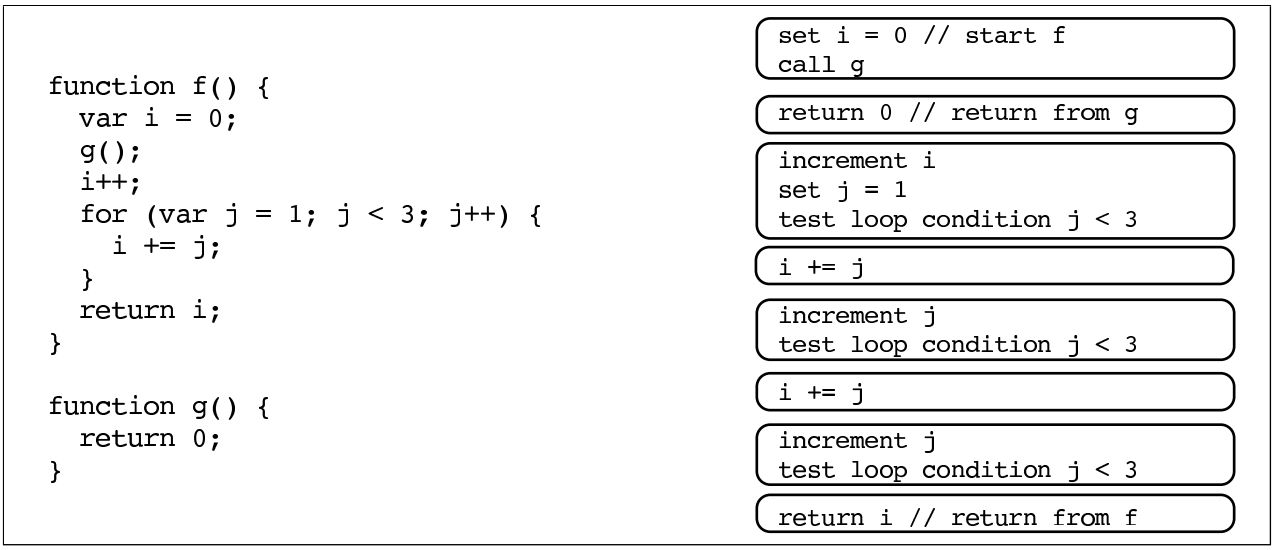


Fig. 1. An example JavaScript function `f` appears on the left. The right shows a high-level, pseudo-intermediate representation of the execution of `f`, to illustrate how tasks are delineated. Each box represents one task. The `i += j` tasks are the loop body tasks.

with the critical path. Since a typical JavaScript opcode in SquirrelFish executes in approximately 30 machine cycles, tasks rather than individual opcodes are a more realistic granularity for parallelization, given the overheads associated with parallelization. We use the procedure detailed in Algorithm 1 to calculate critical paths for the events observed in the dynamic execution traces.

Potential speedup from parallelization is measured by calculating the length of the critical path for each event that was observed in a dynamic execution trace, summing them together, and dividing into the total execution time, that is,

$$Speedup = \frac{T}{\sum_{i=0}^{n-1} c_i} \quad (1)$$

where n is the number of events, c_i is the length of the critical path for event i in cycles, and T is the total number of cycles for the entire execution. This optimistically assumes there is no limit in the number of processors available to execute independent JavaScript tasks.

We conservatively force events to execute in the order that they were observed in the original execution trace. If JavaScript were able to run in parallel, one could imagine a few JavaScript events occurring and running simultaneously, such as a timer firing at the same time as an `onClick` event generated by the user clicking on a link. However, from the interpreter's perspective, it is difficult to distinguish between true user events that must be executed sequentially and automatic events that could be parallelized, making it very challenging to know whether such events could logically be executed simultaneously. Therefore, we do not allow perturbations in *event* execution order.

Additionally, while interpreting some JavaScript functions, some built-in functions in the language make calls to the native runtime, and therefore we cannot observe these instructions executed from the interpreter's perspective. Examples of this function type include the JavaScript `substring`, `parseInt`, `eval`, and `alert` functions. We examine the potential speedup in two cases: the first conservatively assum-

Algorithm 1 Critical Path Calculation for Events

```
{build dependency graph}
for each opcode do
  for each argument in opcode do
    if argument is a write then
      record this opcode address as the most recent writer
    else if argument is a read then
      find last writer to the read location
      if read and last writer are in different tasks, but in
      the same event then
        mark dependency
        update longest path through this task, given the
        new dependency's longest path
      end if
    end if
  end for
  update longest path through this task with this processed
  opcode's cycle count
end for

{find critical path for each event}
for each task in tasks do
  event ← this task's event
  if the longest path through this task > the event's current
  critical path then
    event's critical path ← longest path through task
  end if
end for
```

ing that all calls to the runtime depend on the previous calls; and the second assuming no dependencies ever occur across the runtime. We take these elements into consideration in order to make our parallelism measurements as realistic as possible.

B. Experimental Setup

In order to measure the potential speedup of JavaScript on the web, we instrumented the April 1, 2010 version of the interpreter in SquirrelFish Extreme (WebKit's JavaScript engine, used in Safari and on the iPhone). We then used Safari with our instrumented interpreter to record dynamic execution traces of the JavaScript opcodes executed while interacting with the websites in our benchmark suite.

A growing number of recent studies have concluded that typical JavaScript benchmark suites, such as SunSpider and V8, are not representative of JavaScript behavior on the web [15], [26]. Therefore, we created our own benchmark suite, partly comprised of full interactions with a subset of the top 100 websites from the Alexa Top 500 Websites [18]. We supplemented our benchmark suite with a few other web applications that push the limits of JavaScript engines today, and the V8 version 2 benchmarks for reference. With the exception of the V8 benchmarks that simply run without user input, the length of logged interactions ranged from five to twenty minutes. Table I explains the details of our benchmark collection.

Traces were generated on a 2.4 GHz Intel Core 2 Duo running Mac OS 10.6.3 with 4 GB of RAM. Additionally, we used `RDTSC` instructions to access the Time Stamp Counter to count the number of cycles each JavaScript opcode took to execute. However, the cycle counts that we measured had several confounding factors, resulting in a variance of sometimes several orders of magnitude: the JavaScript interpreter was run in a browser, influencing caching behavior, along with network latency and congestion effects out of our control [27]. Therefore, out of the approximately 400M total dynamic JavaScript opcodes observed, for each type of opcode we selected the minimum cycle count observed and placed these values in a lookup table we used to calculate the critical paths for events. Additionally, if making repeated XML HTTP requests is the primary action of a JavaScript program, the perceived speedup may be less than our measurements because network latency may come to dominate the time that a user is waiting.

Benchmark	Classification	Details
ALEXA TOP 100 WEBSITES		
`bing`	search engine	two textual searches on bing.com and browsed images
`cnn`	news	read several stories on cnn.com
`facebook`	micro data	posted on wall, wrote on other's walls, expanded news feed
`flickr`	photo sharing	visited contact's pages, posted comments
`gmail`	productivity	archived emails, wrote and sent email
`google`	search engine	two textual searches and looked at images on google.com
`googleDocs`	productivity	edited and saved a spreadsheet, shared with others
`googleMaps`	visualization	found a driving route, switched to public transit route, zoomed in to map
`googleReader`	micro data	read several items, mark as unread, share item
`googleWave`	compute intensive	create new wave, add people, type to them, reply to wave
`nyTimes`	news	read several stories on nyTimes.com
`twitter`	micro data	posted tweet, clicked around home page for @replies and lists
`yahoo`	search	perform several searches and look at images
`youtube`	video	watch several videos, expand comments
COMPUTE INTENSIVE SUPPLEMENTAL BENCHMARKS		
`ballPool`	data visualization	chromeexperiments.com/detail/ball-pool/
`fluidSim`	compute intensive	fluid dynamics simulator in JavaScript at nerget.com/fluidSim/
`pacman`	compute intensive	NES emulator written in JavaScript playing Pacman at benfirshman.com/projects/jsnes/
V8 BENCHMARKS VERSION 2		
`v8-crypto`	V8	Encryption and decryption
`v8-deltablue`	V8	One-way constraint solver
`v8-earley-boyer`	V8	Classic Scheme benchmarks, translated to JavaScript by Scheme2Js compiler
`v8-raytrace`	V8	Ray tracer
`v8-richards`	V8	OS kernel simulation

TABLE I
PROVIDES THE DETAILS AND A GENERAL CLASSIFICATION FOR THE BENCHMARKS USED IN THIS STUDY.

IV. RESULTS

In this section, we first evaluate the overall degree of potential speedup. Then we explore other facets of the parallelization problem, including which types of tasks are most useful for maximizing speedup and what types of dependences manifest in typical JavaScript programs.

A. Total Parallelism

The graph in Figure 2 shows the potential speedup, conservatively assuming dependencies across calls to the native runtime, with both function call boundaries and outermost loop bodies as task delimiters. As previously mentioned, runtime calls are a set of built-in functions implemented natively rather than in JavaScript. To reiterate, we conservatively force events to execute in the same order that they were observed in the original execution trace, and we use Equation (1) to calculate the potential speedup. The potential speedups range from 2.19x and 2.31x for the `v8-crypto` and `google` benchmarks respectively, and up to 45.46x on `googleWave`,

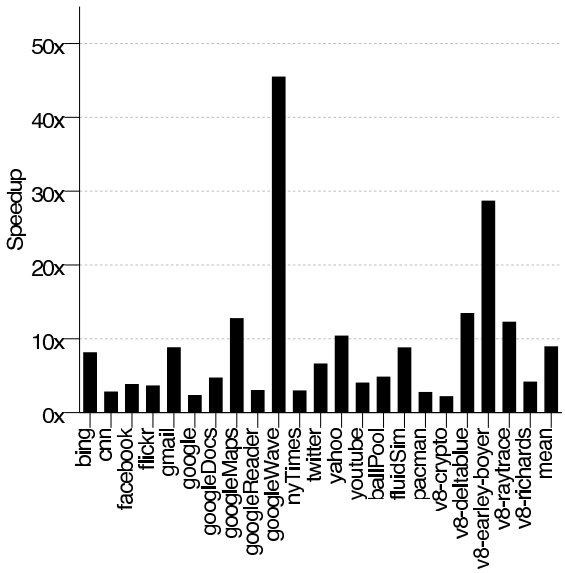


Fig. 2. Potential speedup with runtime dependencies, and function and loop task delimiters.

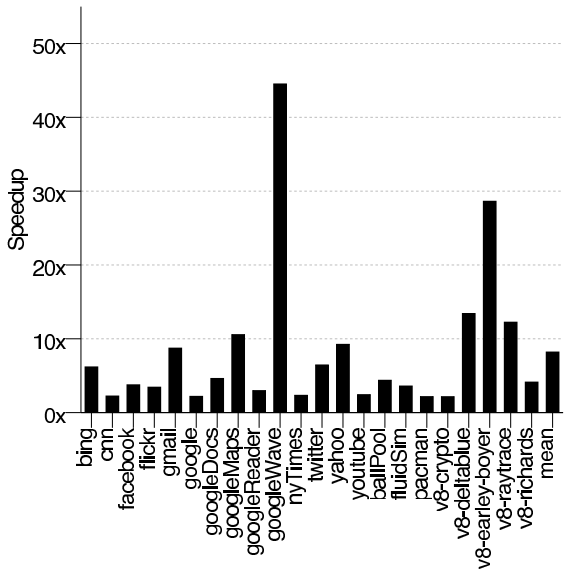


Fig. 3. The potential speedup for only function delimited tasks, with runtime dependencies.

with an average over all the benchmarks of 8.91x. If we exclude the minimum and maximum outliers (`googleWave` and `v8-crypto`, respectively), the adjusted (arithmetic) mean is 7.42x.

B. Task Granularity—Functions and Loops

Figures 3 and 4 illustrate the speedup measurements for function-delimited and outermost loop delimited tasks. We divided up the execution trace into tasks using the method in Figure 1 and discussed in Section III-A. When we break up tasks at loop boundaries, we do not include opcodes related to modifying the loop induction variable in the loop task. However, opcodes that use the loop induction variable to index into an array lookup in the loop body do cause a dependency between the loop task and its parent function task. If JavaScript had explicit support for parallel loop iterations, it is possible that some of these currently unparallelizable loop bodies, such as the array index example, could be rewritten in a way that would enable parallelization.

The graphs indicate that loops alone, in today's JavaScript web applications do not yet seem to be good candidates for automatic loop parallelization. The two leaders in speedup with only loop delimited tasks, `fluidSim`, and `ballPool` were the two most scientific-computing-like benchmarks in the benchmark suite. This does suggest that there may be benefits for loop parallelization in the future, should JavaScript performance improve to the point that developers are willing to write more complex and compute-intensive JavaScript programs.

The CDF in Figure 5 illustrates the iteration counts of each loop sequence in each benchmark, excluding the V8 benchmarks, which do not exhibit the same looping behavior as the "typical" web-site benchmarks. As an example, we count the code in Figure 1 as one distinct loop sequence executing for two iterations. If function f were called again, we would count the loop as a separate loop sequence. We find that 52% of loops in the suite iterated only one or two times. However, we can see a very long tail in the graph; a few loops were very hot—one loop in the `pacman` benchmark executed 1.4M times.

The graphs in Figures 6 and 7 show the frequency with which loops execute for different numbers of iterations in the `fluidSim` and `googleWave` benchmarks. We singled out these two benchmarks because `fluidSim` had the largest speedup with only loop delimited tasks, and `googleWave` had the largest overall speedup. The `googleWave` shows behavior typical of most of the benchmarks: a large number of short executing loops. `fluidSim` is unusual because the dramatic slope increase in the graph is bimodal; in this benchmark most loops execute only a few iterations, but then a large number of loop sequences also loop 4k times.

The two outliers for loop-only delimited tasks, `fluidSim` and `ballPool`, did not prove to have an appreciable difference in the *fraction* of loops executed relative to the entire program size compared to other benchmarks. In fact, we found that the fluid dynamics simulator actually had a *smaller* number of loops executed relative to its program size than a number of the less loop-parallelizable benchmarks. Instead, the average loop body (task) sizes were among the very largest observed in all the benchmarks. This suggests that `fluidSim` and `ballPool` have more long, independent loops, rather than a larger proportion of loop iterations. Although other benchmarks had smaller loop bodies on average, such loop bodies were not often independent from each other.

Task Size. Additionally, we examined the average task size to ensure that our level of granularity is realistic for making

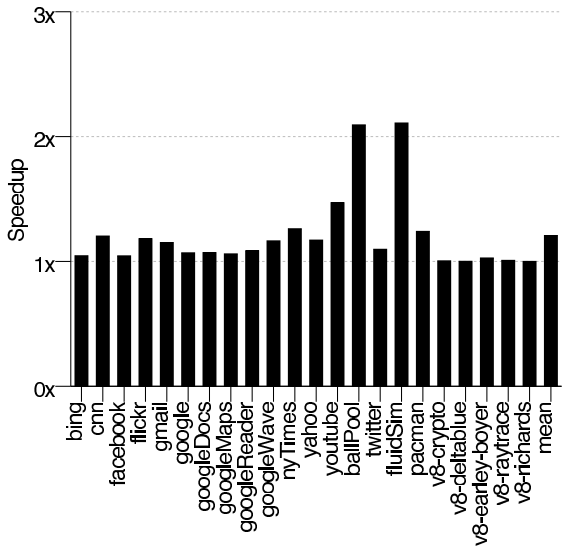


Fig. 4. The potential speedup for only loop delimited tasks, with runtime dependencies.

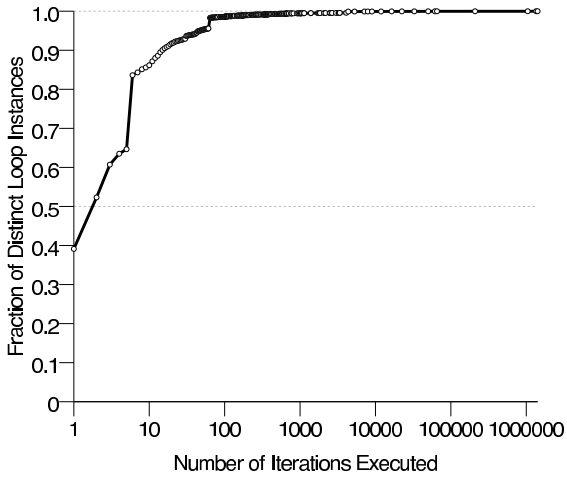


Fig. 5. CDF showing the number of iterations that each distinct loop instance took, incorporating data from all the benchmarks except for the V8 benchmarks. About half of the suite's loop sequences iterate only once or twice.

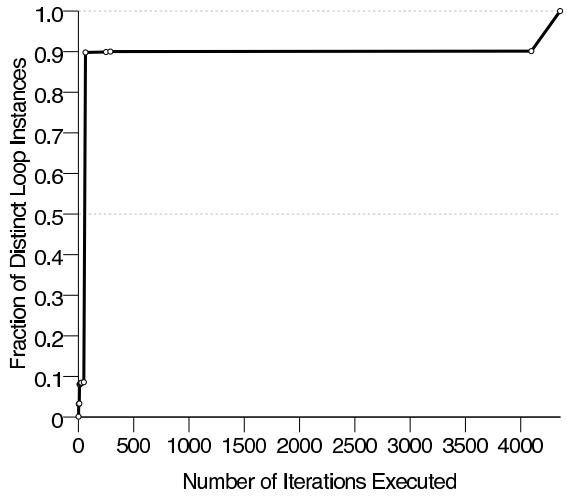


Fig. 6. CDF showing the frequency of loop instances executing for a given number of iterations in the `fluidSim` benchmark. This benchmark has a second large increase in slope in the right side of the graph, indicating a large number of loop instances with a high number of iterations (4k iterations).

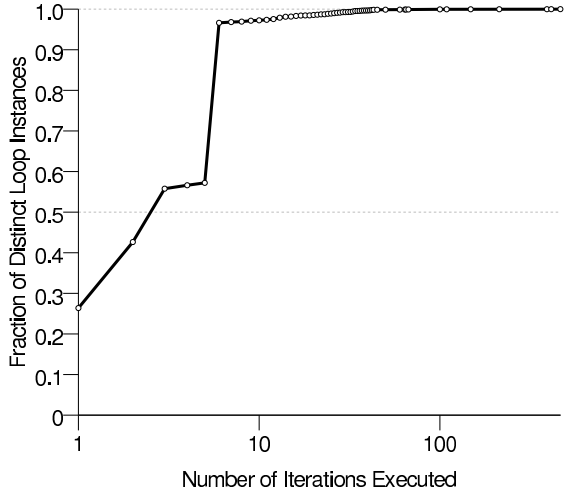


Fig. 7. CDF showing the frequency of loop instances executing for a given number of iterations throughout the `googleWave` benchmark. In this benchmark, the hottest loop instance executes 464 times. The shape of this CDF is very representative of most JavaScript benchmarks.

parallelization worthwhile. As shown in Figure 8, the average task size in most benchmarks is around 17 SquirrelFish opcode instructions, or 650 cycles, with a few outliers with much larger task sizes, such as `v8-crypto` (137 opcodes) and `fluidSim` (73 opcodes). Because task sizes were this short, we see little reason to examine parallelism at an even finer granularity. However, 17 opcodes, or 650 cycles is still quite a fine granularity — we chose this granularity due to the limit study nature of this work, and moreover, tasks can be combined where appropriate during an implementation to better amortize parallelization overheads.

C. Dependences

Below, we further explore the types of dependencies, average dependency lengths, and our assumptions about runtime dependencies in JavaScript. Figure 9 indicates that the majority of dependencies arise from local variables in virtual registers. At least 84% of the dependencies of the web page-based benchmarks (as opposed to the V8 benchmarks) were register-based, which provides promise for static analysis and potentially some automatic and even non-speculative parallelization. Many of the V8 benchmarks, in contrast, had a much higher incidence of memory dependencies.

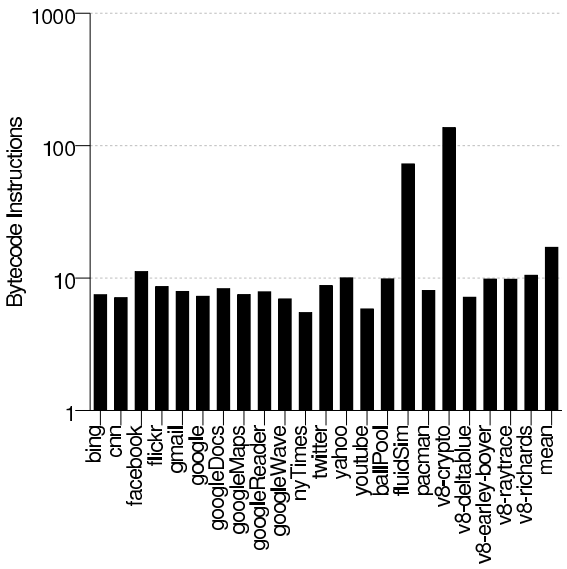


Fig. 8. The average task size of benchmarks by SquirrelFish opcode count, on a log scale.

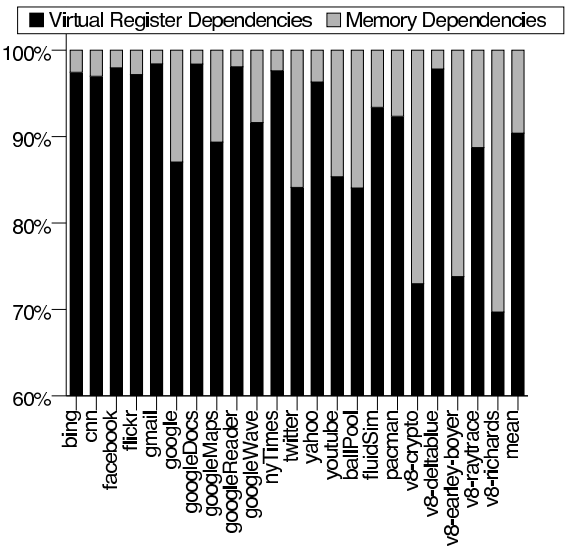


Fig. 9. Percentage of all dependencies that are virtual register to register dependencies *between* tasks versus hash-table lookups between tasks.

Dependency distance refers to the number of cycles (or instructions) between a write to a virtual register or hash table and its subsequent read. Figure 10 shows the average hash table and virtual register dependency distances by JavaScript opcode count. We see that in general, virtual register dependency lengths are shorter than the lengths for memory dependences (average of 50k opcodes for virtual register dependencies versus 160k for hash table lookups). The difference is unsurprising because virtual registers house local and temporary variables, whereas storing data into objects

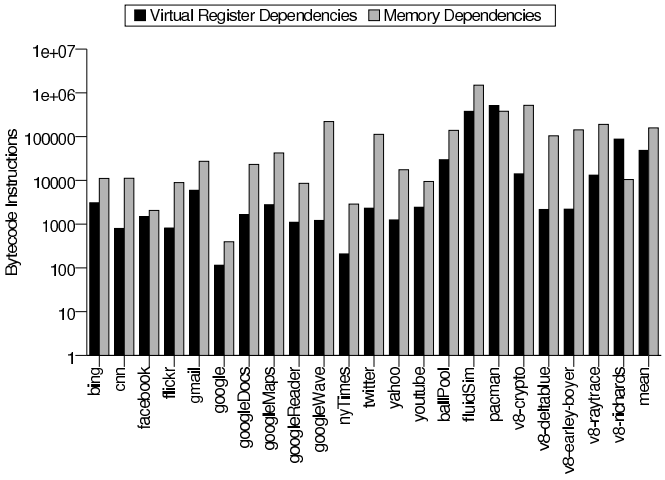


Fig. 10. Average memory and register dependency distances *between* tasks by SquirrelFish opcode count, on a log scale.

that may be referenced from function to function manifest as memory dependences. The graph indicates that the average dependency distance for both virtual registers and hash table lookups is fairly high. Large dependency distances may provide more flexibility for an automatic parallelization, since there is slack to exploit. Additionally, because the memory dependence distance is large, we can more easily amortize the cost of dependence checks.

We find that our more conservative assumption of adding "fake" dependencies[5] into and out of runtime calls did not strongly affect our parallelism measures. Figure 11 shows that although the potential speedups increased slightly from 2.31 to 2.40 for `google` and from 45.46 to 45.89 for `googleWave` without the assumption of runtime dependencies, these numbers are not significantly larger from their more conservative counterparts. This is also encouraging because tracking dependencies via the internal state of the runtime system may be expensive; we can use conservative assumptions without hurting parallelism much.

D. Case Study: googleWave

Finally, we inspected the maximum speedup outlier, `googleWave`, in more detail to better understand what makes the program so parallelizable. We first graphed a Cumulative Distribution Function (CDF) illustrating the frequency of speedup levels of for all *events*, weighted by serial event length, in Figure 12. We see that 75% of all events weighted by execution time have a speedup of 45x or higher, and 50% of all events have a potential speedup of 90x or higher. If we create a similar CDF without weighting events by execution time, we find that 46% of all events have a speedup of 1x and therefore cannot be parallelized. This data indicates that `googleWave` has a large number of very short events that

[5]Since we could not track dependencies called in native code.

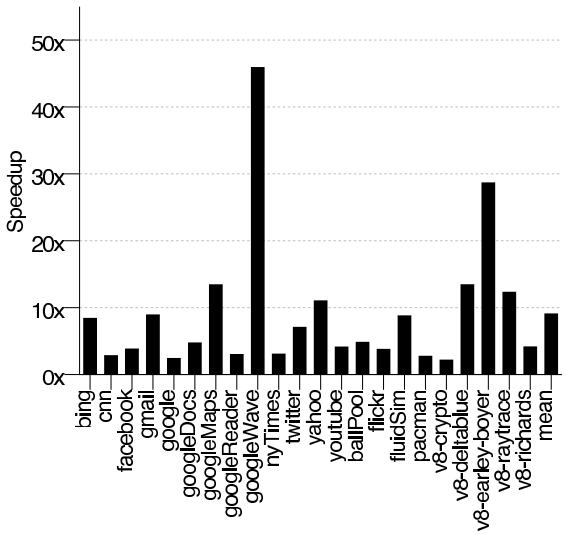


Fig. 11. Potential speedup *without* runtime dependencies, but with function and loop task delimiters

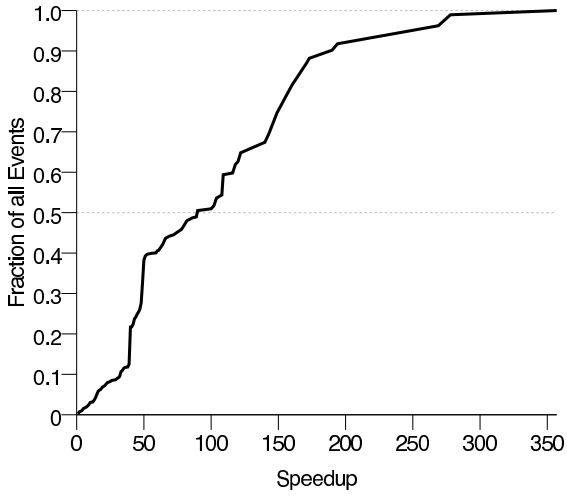


Fig. 12. CDF illustrating the fraction of events that that have a given amount of speedup in the `googleWave` benchmark, weighted by the length of the events. 50% of all events, weighted by event length, have a speedup between 90x and 357x.

will not benefit from parallelization, but a significant number of long running events that are very parallelizable.

We then created a CDF of the size of tasks over the entire program. In Figure 13 we find that 52% of the tasks contain 6 opcodes (180 cycles) or more, and it has a very long tail; the largest task contains 10k opcodes. Today, most web developers are limited by performance reasons from producing applications as JavaScript intensive as `googleWave`.

It should also be noted that `googleWave` is the only benchmark we investigated written using the Google Web Toolkit (GWT). As mentioned in the introduction, this tool compiles Java into optimized JavaScript. It is possible that the

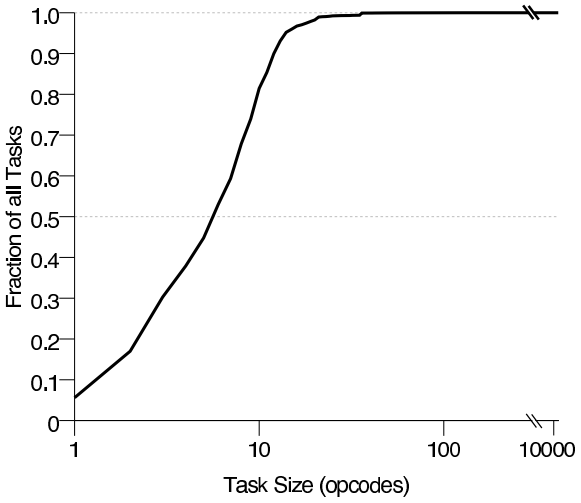


Fig. 13. CDF of task size (in opcodes) in Google Wave. We find over forty percent of all tasks observed in Google Wave are 10 opcodes or more. The largest task in this benchmark contains 10096 opcodes.

coding style of Java lends itself to more short function calls (such as accessor methods in Java) than in JavaScript (which might instead use a hashtable lookup), or GWT automatically generates more parallelizable JavaScript.

Although `googleWave` is something of an outlier amongst other current uses of JavaScript, we believe that in the future we will see a greater number of applications with similar characteristics and similar parallelism opportunities to `googleWave`. There has been a recent trend toward browser-based applications (such as Google Docs, Microsoft's new web-based Office applications, and Chrome OS). To provide a desktop-like experience on the web, such applications would benefit from multi-threading and more performance that could be gained from parallelization.

V. Related Work

JavaScript and its behavior on the web has only recently become a more widely explored topic. Several researchers have explored the behavior of JavaScript from a higher-level language perspective.

Lebresne et al. examined JavaScript objects, how frequently their properties and data are modified, and the length of their prototype chains in the SunSpider benchmarks and a small set of real web pages [17]. This higher-level analysis focused on objects and their modification in order to ultimately provide a type system for JavaScript. Their analysis indicates that the average lengths of object prototype chains across the benchmarks (and also maximum chain lengths for websites) were relatively similar and short, supporting our decision to select the minimum observed cycle value for `op_put_by_val` and `op_get_by_val`.

Ratanaworabhan et al. very extensively measured the usage of JavaScript on the web compared to the V8 and SunSpider benchmarks by instrumenting Internet Explorer 8 and char-

acterizing the behavior of functions, heap-allocated objects and data, and events and handlers [15]. They conclude that the behavior of the "standard" benchmarks is significantly different from actual web pages. Their web page traces were generally based on short web page interactions (*e.g.*, sign in, do one small task, sign out) in comparison to our study's longer running logs. Their results corroborate ours in that in web pages there are a small number of longer-running "hot" functions (that may be worthy of parallelism).

Richards et al. also performed a more extensive analysis of the dynamic behavior of JavaScript usage on the web and compared it to that of standard industry JavaScript benchmarks as well as the conventional wisdom of how JavaScript is used [26]. Whereas Ratanaworabhan et al. used short web interactions, the Richards et al. study recorded longer running traces when other users used the instrumented browser as their default browser. The study investigated, among many things, the frequency of `eval`, the frequency that properties were added to an object after initialization, the variance of the prototype hierarchy, and the frequency of variadic functions. They concluded that the behavior on the web and usage of JavaScript language constructs differed significantly from the prevailing opinion as well as that of the standard benchmarks. While our study analyzes the dynamic behavior of JavaScript on the web and benchmarks arising from low level data dependencies in JavaScript opcodes in order to characterize the potential speedup through parallelization, Richards et al. focuses the analysis primarily at the higher level syntactic layer of JavaScript, with a nod to applications of their results to type systems.

Meyerovich and Bodík find that at least 40% of Safari's time is spent in page layout tasks [16]. They also break down the length of time for several websites spent on various tasks during page loads, of which JavaScript is a relatively small component (15%-20% of page load time). Given these results, they detail several algorithms to enable fast web page layout in parallel. However, *after* page load time, the proportion of time spent executing JavaScript and making XML HTTP requests increases. Indeed, if Meyerovich and Bodík succeed in dramatically reducing the amount of time spent in page loads, then the time spent after page loads, and therefore in JavaScript would increasingly dominate the execution time.

Mickens et al. developed the Crom framework in JavaScript that allows developers to hide latency on web pages by speculatively executing and fetching content that will then be readily available should it be needed [28]. Developers mark specific JavaScript event handlers as speculable, and then the system fetches data speculatively while waiting on the user, reducing the perceived latency of the page.

Meanwhile, some work has been done on the compiler end to improve the execution of JavaScript. Gal et al. implemented TraceMonkey, to identify hot *traces* for the Just-In-Time (JIT) compiler, as opposed to the standard method-based JIT [14]. However, they found that a trace must be executed at least 270 times before they break even from the additional JIT'ing of traces that they do during execution. As we saw in the benchmarks in our study, half of all loops executed in JavaScript programs only execute at most for two iterations. Ha et al. alleviated the performance problem in [14] by building another trace-based Just-In-Time compiler that does its hot trace compilation work in a separate thread so that code execution need not halt to compile a code trace [13]. Martinsen and Grahn have investigated implementing thread level speculation for scripting languages, particularly, JavaScript [12]. Their algorithm achieves near linear speedup for their for-loop intensive test programs. However, as we found in our study, JavaScript as currently used on the web is not written in such a way that its iterations are independent.

VI. Conclusion

Our study is the first exploration of the limits of parallelism within JavaScript applications. We used a collection of traces from popular websites and the V8 benchmarks to observe the data and user-limited dependences in JavaScript. In summary, we found:

a) JavaScript programs on the web hold promise for parallelization: Even without allowing parallelization between JavaScript events, the speedups ranged from 2.19x up to 45.46x, with an average of 8.91x. These numbers are great news for automatic parallelization efforts.

b) Programs are best parallelized within or between functions, not between loop iterations: Most JavaScript loops on the web today do not iterate frequently enough, and iterations are generally not independent enough to be parallelized. Instead, parallelization opportunities arise between tasks formed based on functions. These tasks are large enough to amortize reasonable overheads.

c) The vast majority of data dependences come from virtual registers, and the length of these dependences are shorter than hash-table lookup dependences: This finding makes static analysis conceivable and argues that disambiguation for parallelization can likely be made low cost.

These results suggest that JavaScript holds significant promise for parallelization, reducing execution time and improving responsiveness. Parallelizing JavaScript would enable us to make better use of lower-power multi-core processors on mobile devices, improving power usage and battery life. Additionally, faster JavaScript execution may enable future programmers to write more compute intensive, richer applications.

Acknowledgments

The authors would like to thank Jeff Kilpatrick, Benjamin Lerner, Brian Burg, members of the UW sampa group, and the `#squirrelfish` IRC channel for their helpful discussions.

References

[1] A. Vance, "Revamped Microsoft Office Will Be Free on the Web," *New York Times*, May 11, 2010. [Online]. Available: http://www.nytimes.com/2010/05/12/technology/12soft.html

[2] M. Allen. (2009) Overview of webOS. Palm, Inc. [Online]. Available: http://developer.palm.com/index.php?option=com_content\&view=article\&id=1761\&Itemid=42

[3] (2010, Jun) JavaScript Usage Statistics. BuiltWith. [Online]. Available: http://trends.builtwith.com/javascript
[4] (2010) Google Web Toolkit. Google. [Online]. Available: http://code.google.com/webtoolkit/
[5] F. Loitsch and M. Serrano, "Hop Client-Side Compilation," in *Symposium on Trends on Functional Languages*, 2007.
[6] (2010, June) wave-protocol. [Online]. Available: http://code.google.com/p/wave-protocol/
[7] (2009) w3compiler. Port80 Software. [Online]. Available: http://www.w3compiler.com/
[8] G. Baker and E. Arvidsson. (2010) Let's Make the Web Faster. [Online]. Available: http://code.google.com/speed/articles/optimizing-javascript.html
[9] D. Crockford. (2003, Dec 4,) The JavaScript Minifier. [Online]. Available: http://www.crockford.com/javascript/jsmin.html
[10] (2010, June) Qualcomm Ships First Dual-CPU Snapdragon Chipset. [Online]. Available: http://www.qualcomm.com/news/releases/2010/06/01/qualcomm-ships-first-dual-cpu-snapdragon-chipset
[11] (2010) Mirasol Display Technology. [Online]. Available: http://www.qualcomm.com/products_services/consumer_electronics/displays/mirasol/index.html
[12] J. Martinsen and H. Grahn, "Thread-Level Speculation for Web Applications," in *Second Swedish Workshop on Multi-Core Computing*, Nov 2009.
[13] J. Ha, M. Haghighat, S. Cong, and K. McKinley, "A Concurrent Trace-based Just-in-Time Compiler for JavaScript," in *PESPMA '09: Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, Jun 2009.
[14] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based Just-in-Time Type Specialization for Dynamic Languages," in *PLDI '09: Proceedings of the Conference on Programming Language Design and Implementation*, Jun 2009.
[15] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "JSMeter: Characterizing Real-World Behavior of JavaScript Programs," Microsoft Research, Microsoft Research Technical Report 2009-173, Dec 2009. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=115687
[16] L. Meyerovich and R. Bodik, "Fast and Parallel Webpage Layout," in *WWW '10: Proceedings of the Conference on the World Wide Web*, Apr 2010.
[17] S. Lebresne, G. Richards, J. Östlund, T. Wrigstad, and J. Vitek, "Understanding the Dynamics of JavaScript," in *STOP '09: Proceedings for the Workshop on Script to Program Evolution*, Jul 2009.
[18] (2010, May) Top Sites: The Top 500 Sites on the Web. [Online]. Available: http://www.alexa.com/topsites
[19] (2010, June) V8 Benchmark Suite - version 2. [Online]. Available: http://v8.googlecode.com/svn/data/benchmarks/v2/run.html
[20] (2010, June) What Is SpiderMonkey? [Online]. Available: http://www.mozilla.org/js/spidermonkey/
[21] M. Stachowiak. (2008, September) Introducing SquirrelFish Extreme. [Online]. Available: http://webkit.org/blog/214/introducing-squirrelfish-extreme/
[22] (2010, June) V8 JavaScript Engine. [Online]. Available: http://code.google.com/p/v8/
[23] D. Mandelin. (2008, June) SquirrelFish. [Online]. Available: http://blog.mozilla.com/dmandelin/2008/06/03/squirrelfish/
[24] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual Machine Showdown: Stack Versus Registers," *Transactions on Architecture and Code Optimization*, vol. 4, no. 4, pp. 21:1–21:36, January 2008.
[25] (2010, June) JavaScript Events. [Online]. Available: http://www.w3schools.com/js/js_events.asp
[26] G. Richards, G. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," in *PLDI '10: Proceedings of the Conference on Programming Language Design and Implementation*, Jun 2010.
[27] Microsoft, "Measuring Browser Performance: Understanding Issues in Benchmarking and Performance Analysis," pp. 1–14, 2009. [Online]. Available: http://www.microsoft.com/downloads/details.aspx?displaylang=en\&FamilyID=cd8932f3-b4be-4e0e-a73b-4a373d85146d
[28] J. Mickens, J. Elson, J. Howell, and J. Lorch, "Crom: Faster Web Browsing Using Speculative Execution," in *NSDI '10: Proceedings of Symposium on Networked Systems Design and Implementation*, Apr 2010.

Fidelity and Scaling of the PARSEC Benchmark Inputs

Christian Bienia and Kai Li, Princeton University

***Abstract*—A good benchmark suite should provide users with inputs that have multiple levels of fidelity for different use cases such as running on real machines, register level simulations, or gate-level simulations. Although input reduction has been explored in the past, there is a lack of understanding how to systematically scale input sets for a benchmark suite. This paper presents a framework that takes the novel view that benchmark inputs should be considered approximations of their original, full-sized inputs. It formulates the input selection problem for a benchmark as an optimization problem that maximizes the accuracy of the benchmark subject to a time constraint. The paper demonstrates how to use the proposed methodology to create several simulation input sets for the PARSEC benchmarks and how to quantify and measure their approximation error. The paper also shows which parts of the inputs are more likely to distort their original characteristics. Finally, the paper provides guidelines for users to create their own customized input sets.**

I. Introduction

Computer architects need detailed simulations for their microarchitecture designs. However, simulators are typically many orders of magnitude slower than a real machine. Running a set of benchmark applications with realistic inputs on a simulator will far exceed the design time limits, even using thousands of computers.

To reduce the number of simulated instructions of a benchmark, a commonly used method is to take a subset of the instructions with statistical sampling, which requires a sophisticated mechanism in the simulator [4], [5], [20], [25]. Another method is to reduce the size of the input for a benchmark [14], [22]. Reduced inputs are easy to use with any simulator, but there is no systematic framework for scaling input sets in a continuous way. Furthermore, there is a lack of understanding of the tradeoffs between accuracy and cost involved when reducing inputs for a given benchmark.

We believe a good benchmark suite should provide users with inputs of multiple levels of fidelity, scaling from realistic inputs for execution on real machines down to small inputs for detailed simulations. The key question is how to scale such inputs for a benchmark so that its execution with a scaled-down input produces meaningful performance predictions for computer architects.

This paper presents a framework for the input-scaling problem of a benchmark. We view scaled inputs as approximations of the original, full-sized inputs. As the inputs are scaled down, the benchmark becomes increasingly inaccurate. The question of how to create many derivatives of real inputs with varying size and accuracy motivates the following optimization problem: *Given a time budget, what is the optimal selection of inputs for a set of benchmark programs?* We will show that this optimization problem is the classical *multiple-choice knapsack problem* (MCKP), which is NP-hard. The formulation of the problem allows us to derive the general guidelines for designing multiple inputs with different levels of fidelity for a benchmark suite.

The paper proposes the methodology and implementation of scaled inputs for the Princeton Application Repository for Shared-Memory Computers (PARSEC) [3]. The PARSEC benchmark suite is designed to represent emerging workloads. The current release (version 2.1) consists of 13 multithreaded programs in computer vision, video encoding, physical modeling, financial analytics, content-based search, and data deduplication. We have used the methodology to design six sets of inputs with different levels of fidelity. The first version of PARSEC was released only two years ago. It has been well adopted by the computer architecture community to evaluate multicore designs and multiprocessor systems.

To evaluate the impact of input scaling, we have defined a measure called *approximation error*. Using this measure, we have analyzed several input sets of PARSEC and shown which reduced inputs are more likely to distort the original characteristics of the PARSEC benchmarks. We furthermore analytically derive a scope for each input which defines the range of architectures for which the input can be expected to be reasonably accurate. Such results are helpful for PARSEC users to understand the implications when creating customized inputs for their simulation time budgets.

This paper makes several contributions. First, it presents a novel methodology to analyze how scaled inputs affect the accuracy of a benchmark suite. Second, it formulates the input selection problem for a benchmark as an optimization problem that maximizes the accuracy of a benchmark subject to a time constraint. Third, it describes how the PARSEC inputs have been scaled and shows which parts of the inputs are more likely to distort the original program characteristics. More importantly, it discusses in which situations small input sets might produce highly misleading program behavior. Finally, the paper provides guidelines for users to create their own customized input sets for PARSEC.

II. Input Fidelity

This section introduces the conceptual foundation and terminology to discuss the inaccuracies in benchmark inputs. Our concepts and definitions follow the terms commonly used in the fields of mathematical approximation theory and scientific modeling [1].

The main insight is that inputs for benchmark programs are almost never real program inputs. An example is a server program whose behavior is input dependent, but its input data contains confidential user information that cannot be made publicly available. Also, computationally intensive applications require too much time to complete. For that reason benchmark inputs are typically derived from real program inputs by reducing them in a suitable way. We call the process of creating a range of reduced inputs *input scaling*. This process can be compared to stratified sampling. The idea of stratified sampling is to take samples in a way that leverages knowledge about the sampling population so that the overall characteristics of the population are preserved as much as possible.

Benchmark inputs can be thought of as models of real inputs. Their purpose is to approximate real program behavior as closely as possible, but even if the benchmark program itself is identical to the actual application, some deviations from its real-world behavior must be expected. We call these deviations the *approximation error* because they are unintended side effects which can distort performance measurements in subtle ways. Sometimes very noticeable errors can be identified. We will refer to these errors as *scaling artifacts*. A scaling artifact can increase the inaccuracy of a benchmark and sometimes it may lead to highly misleading results. In such cases, scaling artifacts are good indicators whether an input set can accurately

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

approximate the real workload inputs. Identifying such artifacts is essential to determine the constraints of a benchmark input set.

We define *fidelity* as the degree to which a benchmark input set produces the same program behavior as the real input set. An input set with high fidelity has a small approximation error with few or no identifiable scaling artifacts, whereas an input set with low fidelity has a large approximation error and possibly many identifiable scaling artifacts. Input fidelity can be measured by quantifying the approximation error. It is common that approximations are optimized to achieve a high degree of fidelity in a limited target area, possibly at the expense of reduced accuracy in other areas. We call this target area of high fidelity the *scope* of an input. The scope can be thought of as a set of restrictions for a reduced input beyond which the input becomes very inaccurate. An example is an overall reduction of work units in an input set, which will greatly reduce execution time, but limit the amount of CPUs that the program can stress simultaneously. Running the benchmark on CMPs with more cores than the input includes may lead to many noticeable scaling artifacts in the form of idle CPUs.

A. Optimal Input Selection

Input scaling can be used to create a continuum of approximations of a real-world input set with decreasing fidelity and instruction count. This allows benchmark users to trade benefit in the form of benchmarking accuracy for lower benchmarking cost as measured by execution or simulation time. This motivates the question what the optimal combination of inputs from a given selection of input approximations is that maximizes the accuracy of the benchmark subject to a time budget.

If the time budget is fixed and an optimal solution is desired, the problem assumes the structure of the multiple-choice knapsack problem (MCKP), which is a version of the binary knapsack problem with the addition of disjoint multiple-choice constraints [15], [21]. For a given selection of m benchmark programs with disjoint sets of inputs N_i, $i = 1,\ldots,m$, we will refer to the approximation error of each respective input $j \in N_i$ with the variable $a_{ij} \geqq 0$. If we define a solution variable $x_{ij} \in \{0,1\}$ for each input that specifies whether an input has been selected then we can formally define the goal of the optimization problem as follows:

$$\min \sum_{i=1}^{m} \sum_{j \in N_i} a_{ij} x_{ij} \tag{1}$$

The solution of the problem is the set of all x_{ij} that describes which inputs are optimal to take. If we refer to the time each input takes to execute with $t_{ij} \geqq 0$ and to the total amount of time available for benchmark runs with $T \geqq 0$ then we can describe which solutions are acceptable as follows:

$$\sum_{i=1}^{m} \sum_{j \in N_i} t_{ij} x_{ij} \leqq T \tag{2}$$

Lastly, we need to state that exactly one input must be selected for each benchmark program and that we do not allow partial or negative inputs:

$$\sum_{j \in N_i} x_{ij} = 1 \quad i = 1,\ldots,m \tag{3a}$$

$$x_{ij} \in \{0,1\} \quad j \in N_i, \quad i = 1,\ldots,m \tag{3b}$$

MCKP is an NP-hard problem, but it can be solved in pseudo-polynomial time through dynamic programming [6], [19]. The most efficient algorithms currently known first solve the linear version of MCKP (LMCKP) that is obtained if the integrality constraint $x_{ij} \in \{0,1\}$ is relaxed to $0 \leqq x_{ij} \leqq 1$. LMCKP is a variant of of the fractional knapsack problem and can be solved in $O(n)$ time by a greedy algorithm. The initial feasible solution that is obtained that way is used as a starting point to solve the more restrictive MCKP version of the problem with a dynamic programming approach. It keeps refining the current solution in an enumerative fashion by adding new classes until an optimal solution has been found. Such an algorithm can solve even very large data instances within a fraction of a second in practice, as long as the input data is not strongly correlated [19].

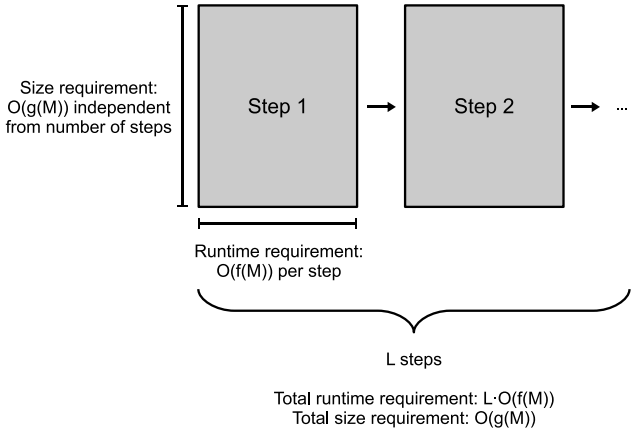


Fig. 1. The impact that linear and complex input size scaling typically have on a program. Complex scaling of M usually affects the execution time $f(M)$ and the memory requirement $g(M)$. Linear scaling typically only changes the number of repetitions L.

B. Scaling Model

The most common way to express the asymptotic runtime behavior of a program is to use a function of a single parameter N, which is the total size of the input. This is too simplistic for real-world programs, which usually have inputs that are described by several parameters, each of which may significantly affect the behavior of the programs. This section presents a simple scaling model for reasoning about the effect of input parameters on program behavior.

In our simple model, we group the inputs of a program into two components: the linear component and the complex component. The linear component includes the parts that have a linear effect on the execution time of the program such as streamed data or the number of iterations of the outermost loop of the workload. The complex component includes the remaining parts of the input which can have any effect on the program. For example, a fully defined input for a program that operates on a video would be composed of a complex part that determines how exactly to process each frame and a linear part that determines how many frames to process.

Our motivation for this distinction is twofold: First, real-world workloads frequently apply complex computational steps which are difficult to analyze in a repetitive fashion that is easy to count and analyze. Second, the two components have different qualitative scaling characteristics because the complex

TABLE I
THE SIX STANDARDIZED INPUT SETS OFFERED BY PARSEC LISTED IN ORDER OF INCREASING SIZE. LARGER INPUT SETS GUARANTEE THE SAME PROPERTIES OF ALL SMALLER INPUT SETS. TIME IS APPROXIMATE SERIAL EXECUTION TIME ON REAL MACHINES.

Input Set	Description	Time Budget T	Purpose
`test`	Minimal execution time	N/A	Test & Development
`simdev`	Best-effort code coverage of real inputs	N/A	
`simsmall`	Small-scale experiments	$\lessapprox$ 1 s	Simulations
`simmedium`	Medium-scale experiments	$\lessapprox$ 4 s	
`simlarge`	Large-scale experiments	$\lessapprox$ 15 s	
`native`	Real-world behavior	$\lessapprox$ 15 min	Native execution

input part often has to remain in a fairly narrow range whereas the linear component can often be chosen rather freely. In the example of the video processing program, the complex input part determining the work for each single frame is limited by the algorithms that are available and need to be within the range of video frame sizes that make sense in practice, which means there is a tight upper bound on the number of processing steps as well as lower and upper bounds on the frame resolution that make sense. These bounds significantly limit the scaling range, and the possible choice among different algorithms make this part of the input difficult to analyze. The number of frames, however, is unconstrained and can practically reach any number.

These two components of an input affect the input scaling behavior of the program in different ways, as summarized in Figure 1. The impact of the complex input parts is typically highly dependent on the workload so that few general guidelines can be given. It is common that the asymptotic runtime of the program increases superlinearly but its memory requirements often grow only linearly because most programs only keep the input data in a parsed but essentially unmodified form in memory. This property makes it hard to scale an input using its complex components without skewing it. Complex scaling almost always reduces working set sizes.

The linear input components typically do not affect the memory requirements or working set sizes of the program that much because they involve a form of repetition of previous steps. Their strong and direct impact on the execution time of the program make them suitable for input size scaling. Usually, there is no upper limit for linear input scaling, but reducing the input to the point where it includes few if any repetitions can often result in strong scaling artifacts because the individual steps are not exactly the same or not completely independent from each other. For example, if a program takes the output of its previous iteration as the input for its next iteration, it usually results in significant communication between threads. The underlying communication patterns may vary significantly from one iteration to the next. In this case, the number of repetitions included in the input must be large enough to stabilize what type of tasks the program performs on average.

III. PARSEC INPUTS

This section describes the inputs of PARSEC and how they are scaled. PARSEC offers six input sets. Each set contains exactly one fully defined input for each PARSEC benchmark. An input is composed of all input files required by the program and a predetermined way to invoke the binary. Input sets can be distinguished by the amount of work their inputs contain. This determines what an input set can be used for. Table I gives an overview of the six PARSEC input sets ordered in ascending order by the allowed time budget.

The `native` input set is the closest approximations to realistic inputs, even though it is not authoritative. Only benchmarks with real-world programs using real-world inputs are authoritative [12]. Smaller input sets can be considered increasingly inaccurate approximations of real-world inputs. Users of the benchmark suite should therefore generally use the largest input set possible. The smallest input set which we consider acceptable for at least some performance experiments is `simsmall`.

A. Scaling of PARSEC Inputs

PARSEC inputs predominantly use linear input scaling to achieve the large size reduction from real inputs to `native` and `simlarge` and a combination of linear and complex scaling to derive the `simmedium` and `simsmall` input sets from `simlarge`. For that reason the differences between real inputs, `native` and `simlarge` should be relatively small. The input sets `simdev` and `test` were created in a completely different way and should not be used for performance experiments at all. The various inputs suitable for performance measurements are summarized in Table II.

Most parts of the complex input components are identical between the input sets `simlarge` and `native`. In seven cases at least one part of the complex input component is not identical between the two input sets: `Canneal`, `ferret`, `fluidanimate`, `freqmine`, `raytrace`, `streamcluster` and `x264` all have at least one input component that exhibits complex scaling behavior which might affect the program noticeably. However, only in the case of `streamcluster` could we measure a noticeable and strong impact of that property on the program characteristics. This is because there is a direct, linear relationship between the working set of the program and the selected block size, which can be freely chosen as part of the input and which has also been scaled between input sets. For `simlarge` the working set corresponding to the block size is 8 MB, which means a working set size between 64 MB and 128 MB can be expected for the `native` input. In all other cases we expect the differences between `simlarge` and `native` to be negligible on contemporary machines.

`Blackscholes`, `dedup`, `swaptions` and `vips` all break their input into small chunks which are processed one after another. Their inputs are the easiest to scale and generally should show little variation. In the case of `blackscholes` and `dedup` some impact on the working set sizes should be expected because each input unit can be accessed more than once by the program.

The most difficult inputs to scale are the ones of `freqmine`. They exhibit no linear component, which means that any form of input scaling might alter the characteristics of the workload significantly. Moreover, `freqmine` parses and stores its input data internally as a frequent-pattern tree (FP-tree) that will be mined during program execution. An FP-tree is a compressed form of the transaction database that can be traversed in multiple ways. This makes the program behavior highly dependent on the exact properties of the input data, which might further amplify the problem.

TABLE II
OVERVIEW OF PARSEC INPUTS AND HOW THEY WERE SCALED. WE POINT OUT IN WHICH CASES THE EXACT CONTENTS OF THE INPUT DATA CAN HAVE A STRONG IMPACT ON THE CODE PATH OR THE CHARACTERISTICS OF THE PROGRAM.

Program	Input Set	Problem Size		Comments
		Complex Component	Linear Component	
blackscholes	simsmall		4,096 options	
	simmedium		16,384 options	
	simlarge		65,536 options	
	native		10,000,000 options	
bodytrack	simsmall	4 cameras, 1,000 particles, 5 layers	1 frame	
	simmedium	4 cameras, 2,000 particles, 5 layers	2 frames	
	simlarge	4 cameras, 4,000 particles, 5 layers	4 frames	
	native	4 cameras, 4,000 particles, 5 layers	261 frames	
canneal	simsmall	100,000 elements	10,000 swaps per step, 32 steps	
	simmedium	200,000 elements	15,000 swaps per step, 64 steps	
	simlarge	400,000 elements	15,000 swaps per step, 128 steps	
	native	2,500,000 elements	15,000 swaps per step, 6,000 steps	
dedup	simsmall		10 MB data	Data affects behavior
	simmedium		31 MB data	
	simlarge		184 MB data	
	native		672 MB data	
facesim	simsmall	80,598 particles, 372,126 tetrahedra	1 frame	Complex scaling challenging
	simmedium	80,598 particles, 372,126 tetrahedra	1 frame	
	simlarge	80,598 particles, 372,126 tetrahedra	1 frame	
	native	80,598 particles, 372,126 tetrahedra	100 frames	
ferret	simsmall	3,544 images, find top 10 images	16 queries	
	simmedium	13,787 images, find top 10 images	64 queries	
	simlarge	34,793 images, find top 10 images	256 queries	
	native	59,695 images, find top 50 images	3,500 queries	
fluidanimate	simsmall	35,000 particles	5 frames	
	simmedium	100,000 particles	5 frames	
	simlarge	300,000 particles	5 frames	
	native	500,000 particles	500 frames	
freqmine	simsmall	250,000 transactions, min support 220		Data affects behavior
	simmedium	500,000 transactions, min support 410		
	simlarge	990,000 transactions, min support 790		
	native	250,000 transactions, min support 11,000		
raytrace	simsmall	480×270 pixels, 1 million polygons	3 frames	Data affects behavior
	simmedium	960×540 pixels, 1 million polygons	3 frames	
	simlarge	$1,920 \times 1,080$ pixels, 1 million polygons	3 frames	
	native	$1,920 \times 1,080$ pixels, 10 million polygons	200 frames	
streamcluster	simsmall	4,096 points per block, 32 dimensions	1 block	
	simmedium	8,192 points per block, 64 dimensions	1 block	
	simlarge	16,384 points per block, 128 dimensions	1 block	
	native	200,000 points per block, 128 dimensions	5 blocks	
swaptions	simsmall		16 swaptions, 5,000 simulations	
	simmedium		32 swaptions, 10,000 simulations	
	simlarge		64 swaptions, 20,000 simulations	
	native		128 swaptions, 1,000,000 simulations	
vips	simsmall		$1,600 \times 1,200$ pixels	
	simmedium		$2,336 \times 2,336$ pixels	
	simlarge		$2,662 \times 5,500$ pixels	
	native		$18,000 \times 18,000$ pixels	
x264	simsmall	640×360 pixels	8 frames	Data affects behavior
	simmedium	640×360 pixels	32 frames	
	simlarge	640×360 pixels	128 frames	
	native	$1,920 \times 1,080$ pixels	512 frames	

The complex input components of the facesim inputs have not been scaled at all. Doing so would require generating a new face mesh, which is a challenging process. Significant reductions of the mesh resolution can also cause numerical instabilities. The three simulation inputs of facesim are therefore identical and should be considered as belonging to the simlarge input set.

Besides freqmine three more programs significantly alter their behavior depending on the data received. Dedup builds a database of all unique chunks that are encountered in the input stream. Less redundancy in the input stream will cause larger working sets. Raytrace follows the path of light rays through a scene. Small alterations of the scene or the movement of the camera might cause noticeable changes of the execution or working set sizes. However, in natural scenes with realistic camera movement this effect is likely to be small if the number of light rays is sufficiently large because their fluctuations will average out due to the law of large numbers. Finally, x264 uses a significantly larger frame size for its native input. Just like vips the program breaks an input frame into smaller chunks of fixed size and processes them one at a time. However, x264 must keep some frames in memory after they have been processed because it references them to encode subsequent frames. This property increases working set sizes and the amount of shared data with the frame size and is the reason why we classify the image resolution of the input as a complex input component.

B. General Scaling Artifacts

One common scaling artifact caused by linear input scaling is an exaggerated warmup effect because the startup cost has to be amortized within a shorter amount of time. Furthermore, the serial startup and shutdown phases of the program will also appear inflated in relation to the parallel phase. This is an inevitable consequence if workloads are to use inputs with working sets comparable real-world inputs but with a significantly reduced execution time - the programs will have

to initialize and write back a comparable amount of data but will do less work with it.

Consequently, all characteristics will be skewed towards the initialization and shutdown phases. In particular the maximum achievable speedup is limited due to Amdahl's Law if the whole execution of the program is taken into consideration. It is important to remember that this does not reflect real program behavior. The skew should be compensated for by either excluding the serial initialization and shutdown phases and limiting all measurements to the Region-of-Interest (ROI) of the program, which was defined to include only the representative parallel phase, or by measuring the phases of the program separately and manually weighing them correctly. We believe it is safe to assume that the serial initialization and shutdown phases are negligible in the real inputs, which allows us to completely ignore them for experiments. Benchmark users who do not wish to correct measurements in such a way should limit themselves to the `native` input set, which is a much more realistic description of real program behavior that exhibits these scaling artifacts to a much lesser extent.

C. Scope of PARSEC Inputs

In this section we will briefly describe what the constraints of the PARSEC simulation inputs are and under which circumstances we can expect to see additional, noticeable scaling artifacts. The most severe limitations are the amount of parallelism and the size of the working sets.

The PARSEC simulation inputs were scaled for machines with up to 64 cores and with up to tens of megabytes of cache. These limitations define the scope of these inputs, with smaller inputs having even tighter bounds. If the inputs are used beyond these restrictions noticeable scaling artifacts such as idle CPUs caused by limited parallelism must be expected. The `native` input set should be suitable for machines far exceeding these limitations.

TABLE III
WORK UNITS CONTAINED IN THE SIMULATION INPUTS. THE NUMBER OF WORK UNITS PROVIDED BY THE INPUTS IS AN UPPER BOUND ON THE NUMBER OF THREADS THAT CAN WORK CONCURRENTLY. ANY OTHER BOUNDS ON PARALLELISM THAT ARE LOWER ARE GIVEN IN PARENTHESES.

Program	Input Set		
	simsmall	**simmedium**	**simlarge**
blackscholes	4,096	16,384	65,536
bodytrack	60	60	60
canneal	$\leq 5.0 \cdot 10^4$	$\leq 1.0 \cdot 10^5$	$\leq 2.0 \cdot 10^5$
dedup	2,841	8,108	94,130
facesim	80,598	80,598	80,598
ferret	16	64	256
fluidanimate	$\leq 3.5 \cdot 10^4$	$\leq 1.0 \cdot 10^5$	$\leq 3.0 \cdot 10^5$
freqmine	23	46	91
raytrace	1,980	8,040	32,400
streamcluster	4,096	8,192	16,384
swaptions	16	32	64
vips	475 (50)	1369 (74)	3612 (84)
x264	8	32	128

Reducing the size of an input requires a reduction of the amount of work contained in it which typically affects the amount of parallelism in the input. We have summarized the number of work units in each simulation input set in Table III. This is an upper bound on the number of cores that the input can stress simultaneously.

Workloads with noticeably low amounts of work units are `bodytrack`, `ferret`, `freqmine`, `swaptions` and `x264`. `Ferret` processes image queries in parallel, which means that the number of queries in the input limits the amount of cores it can use. This can be as little as 16 for `simsmall`. The amount of parallelism in the simulation input sets of `swaptions` is comparable. The smallest work unit for the program is a single swaption, only 16 of which are contained in `simsmall`. `X264` uses coarse-grain parallelism that assigns whole frames to individual threads. The number of possible cores the program can use is thus restricted to the number of images in the input, which is only eight in the case of `simsmall`. The number of work units that `vips` can simultaneously process is technically limited to the cumulative size of the output buffers, which can be noticeable on larger CMPs. This limitation has been removed in later versions of `vips` and will probably disappear in the next version of PARSEC. The amount of parallelism contained in the `bodytrack` inputs is limited by a vertical image pass during the image processing phase. It was not artificially introduced by scaling, real-world inputs exhibit the same limitation. It is therefore valid to use the simulation inputs on CMPs with more cores than the given limit. The upper bound introduced by input scaling is given by the number of particles, which is nearly two orders of magnitude larger. The natural limitation of parallelism during the image processing phase should only become noticeable on CMPs with hundreds of cores because image processing takes up only a minor part of the total execution time. The bound on parallelism for `canneal` and `fluidanimate` is probabilistic and fluctuates during runtime. It is guaranteed to be lower than the one given in Table III but should always be high enough even for extremely large CMPs.

Input scaling can also have a noticeable effect on the working sets of a workload and some reduction should be expected in most cases. However, the impact is significant in the cases of 'unbounded' workloads [3], which are `canneal`, `dedup`, `ferret`, `freqmine` and `raytrace`. A workload is unbounded if it has the qualitative property that its demand for memory and thus working sets is not limited in practice. It should therefore never fully fit into a conventional cache. Any type of input that requires less than all of main memory must be considered scaled down. For example, `raytrace` moves a virtual camera through a scene which is then visualized on the screen. Scenes can reach any size, and given enough time each part of it can be displayed multiple times and thus create significant reuse. This means the entire scene forms a single, large working set which can easily reach a size of many gigabytes.

A scaled-down working set should generally not fit into a cache unless its unscaled equivalent would also fit. Unfortunately larger working sets also affect program behavior on machines with smaller caches because the miss rate for a given cache keeps growing with the working set if the cache cannot fully contain it. We therefore do not give exact bounds on cache sizes as we did with parallelism, an impact on cache miss rates must be expected for all cache sizes. Instead we will account for this effect by including the cache behavior for a range of cache sizes in the aproximation error.

IV. VALIDATION OF PARSEC INPUTS

This section addresses the issue of accuracy of smaller input sets. We will first describe our methodology and then report the approximation error of the input sets relative to the entire PARSEC benchmark suite.

A. Methodology

An ideal benchmark suite should consist of a diverse selection of representative real-world programs with realistic inputs. As described in Section II, an optimal selection of inputs should minimize the deviation of the program behavior for a target time

limit while maintaining the diversity of the entire benchmark suite. Thus, analyzing and quantifying program behavior is the fundamental challenge in measuring differences between benchmarks and their inputs.

We view program behavior as an abstract, high-dimensional feature space of potentially unlimited size. It manifests itself in a specific way such that it can be measured in the form of characteristics when the program is executed on a given architecture. We can think of this process as taking samples from the behavior space at specific points defined by a particular architecture-characteristic pair. Given enough samples an image of the program behavior emerges.

We chose a set of characteristics and measured them for the PARSEC simulation inputs on a particular architecture. We then processed the data with Principal Component Analysis (PCA) to automatically eliminate highly correlated data. The result is a description of the program and input behavior that is free of redundancy.

We define *approximation error* as the dissimilarity between different inputs for the same program, as mentioned in Section II. One can measure it by computing the pairwise distances of the data points in PCA space. We will use approximation error as the basic unit to measure how accurate a scaled input is for its program. To visualize the approximation error of all benchmark inputs we use a dendrogram which shows the similarity or dissimilarity of the various inputs with respect to each other.

This methodology to analyze program characteristics is the common method for similarity analysis, but its application to analyze approximation errors is new. Measuring characteristics on an ideal architecture is frequently used to focus on program properties that are inherent to the algorithm implementation and not the architecture [2], [3], [24]. PCA and hierarchical clustering have been in use for years as an objective way to quantify similarity [9], [10], [13], [18], [23].

1) Program Characteristics: For our analysis of the program behavior we chose a total of 73 characteristics that were measured for each of the 39 simulation inputs of PARSEC 2.1, yielding a total of 2,847 sample values that were considered. Our study focuses on the parallel behavior of the multithreaded programs relevant for studies of CMPs. The characteristics we chose encode information about the instruction mix, working sets and sharing behavior of each program as follows:

- **Instruction Mix** 25 characteristics that describe the breakdown of instruction types relative to the total amount of instructions executed by the program
- **Working Sets** 8 characteristics encoding the working set sizes of the program by giving the miss rate for different cache sizes
- **Sharing** 40 characteristics describing how many lines of the total cache are shared and how intensely the program reads or writes shared data

The working set and sharing characteristics were measured for a total of 8 different cache sizes ranging from 1 MBytes to 128 MBytes to include information about a range of possible cache architectures. This approach guarantees that unusual changes in the data reuse behavior due to varying cache sizes or input scaling are captured by the data. The range of cache sizes that we considered has been limited to realistic sizes to make sure that the results of our analysis will not be skewed towards unrealistic architectures.

2) Experimental Setup: To collect the characteristics of the input sets we simulate an ideal machine that can complete all instructions within one cycle using `Simics`. We chose an ideal machine architecture because we are interested in properties inherent to the program, not in characteristics of the underlying architecture. The binaries which we used are the official precompiled PARSEC 2.1 binaries that are publicly available on the PARSEC website. The compiler used to generate the precompiled binaries was `gcc 4.4.0`.

We simulated an 8-way CMP with a single cache hierarchy level that is shared between all threads. The cache is 4-way associative with 64 byte lines. The capacity of the cache was varied from 1 MB to 128 MB to obtain information about the working set sizes with the corresponding sharing behavior. Only the Region-of-Interest (ROI) of the workloads was characterized.

3) Principal Component Analysis: Principal Component Analysis (PCA) is a mathematical method to transform a number of possibly correlated input vectors into a smaller number of uncorrelated vectors. These uncorrelated vectors are called the principal components (PC). We employ PCA in our analysis because PCA is considered the simplest way to reveal the variance of high-dimensional data in a low dimensional form.

To compute the principal components of the program characteristics, the data is first mean-centered and normalized so it is comparable with each other. PCA is then used to reduce the number of dimensions of the data. The resulting principal components have decreasing variance, with the first PC containing the most amount of information and the last one containing the least amount. We use the Kaiser's Criterion to eliminate PCs which do not contain any significant amount of information in an objective way. Only the top PCs with eigenvalues greater than one are kept, which means that the resulting data is guaranteed to be uncorrelated but to still contain most of the original information.

4) Approximation Error: All PARSEC inputs are scaled-down versions of a single real-world input, which means the more similar an inputs is to a bigger reference input for the same program the smaller is its approximation error. After the data has been cleaned up with PCA, this similarity between any two PARSEC inputs can be measured in a straightforward manner by calculating the Euclidean distance (or L_2) between them. Inputs with similar characteristics can furthermore be grouped into increasingly bigger clusters with hierarchical clustering. This method assigns each input set to an initial cluster. It then merges the two most similar clusters repeatedly until all input sets are contained in a single cluster. The resulting structure can be visualized with a dendrogram.

Inputs which merge early in the dendrogram are very similar. The later inputs merge the more dissimilar they are. Inputs for the same workload which preserve its characteristics with respect to other programs in the suite should fully merge before they join with inputs from any other benchmarks. Ideally all inputs for the same workload form their own complete clusters early on before they merge with clusters formed by inputs of any other workloads.

5) Limitations: We express all approximation errors relative to `simlarge`, the largest simulation input. While it would be interesting to know how `simlarge` compares to `native` or even real-world inputs, this information is of limited practical value because it is infeasible to use even bigger inputs for most computer architecture studies. For the most part, benchmark users are stuck with inputs of fairly limited size no matter how big their approximation error is. Time constraints also limit the scope of this study because we believe the more detailed behavior space exploration which becomes possible with simulation

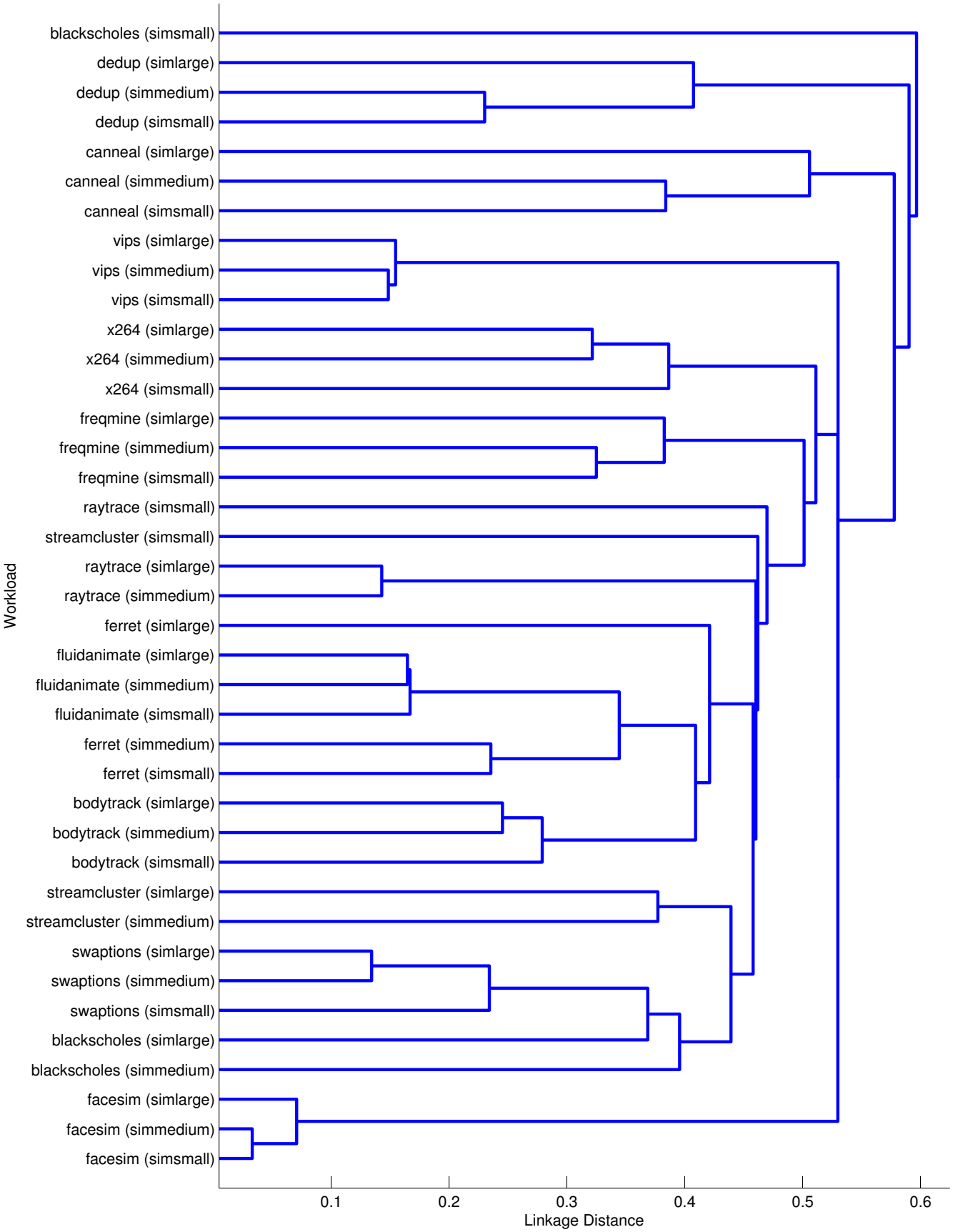


Fig. 2. Fidelity of the PARSEC input sets. The figure shows the similarity of all inputs. The approximation error is the lack of similarity between inputs for the same workload.

is more important than a more accurate reference point which would be feasible with experiments on real machines.

B. Validation Results

We studied the chosen characteristics of the inputs with PCA. The results of this comparison are summarized by the dendrogram in Figure 2.

The dendrogram shows that the inputs of most workloads form their own, complete clusters before they merge with inputs of other programs. That means that the inputs preserve the characteristics of the program with respect to the rest of the suite. These workloads are `dedup`, `canneal`, `vips`, `x264`, `freqmine`, `fluidanimate`, `bodytrack`, `swaptions` and `facesim`.

The inputs for the benchmarks `blackscholes`, `raytrace`, `streamcluster` and `ferret` merge with clusters formed by inputs of other programs before they can form their own, complete cluster. This indicates that the input scaling process skewed the inputs in a way that made the region in the characteristics space that corresponds to the workload overlap with the characteristics space of a different benchmark. This is somewhat less of an issue for `ferret`, which simply overlaps with `fluidanimate` before its inputs can fully merge. However, three inputs belonging to the `simsmall` input set merge significantly later than all other inputs. These are the `simsmall` inputs for `blackscholes`, `raytrace` and `streamcluster`. This indicates that these inputs not only start to be atypical for their workloads, they even become atypical for the entire suite. It is important to emphasize that this increase in diversity is not desirable because it is an artificial byproduct of input scaling which does not represent real-world program behavior. In the case of `streamcluster` the working sets change significantly as its inputs are scaled, as we explained in the last section.

It is also worth mentioning which inputs merge late. While forming their own clusters first, the inputs within the clusters for `dedup`, `canneal`, `x264` and `freqmine` have a distance to each other in the dendrogram which is larger than half of the maximum distance observed between any data points for the entire suite. In these cases there is still a significant amount of difference between inputs for the same workload, even though the inputs as a whole remain characteristic for their program.

The same data is also presented in Figure 3. It shows directly what the distances between the three simulation inputs for each workload are without considering other inputs that might be in the same region of the PCA space. The figure shows the approximation error a_{ij} of the simulation inputs, which is simply the distance from `simlarge` in our case. We have explained that we chose `simlarge` as our reference point because it typically is the best input that is feasible to use for microarchitectural simulations. It is worth mentioning that the data in Figure 3 follows the triangle inequality - the cumulative distance from `simsmall` to `simmedium` and then from `simmedium` to `simlarge` is never less than the distance from `simsmall` to `simlarge`.

The figure shows that the inputs for `bodytrack`, `dedup`, `facesim`, `fluidanimate`, `swaptions` and `vips` have an approximation error that is below average, which means they have a high degree of fidelity. This list includes nearly all benchmarks whose inputs could be scaled with linear input scaling. The inputs with the highest fidelity are the ones for `facesim`. This is not surprising considering that the simulation inputs for that workload have not been scaled at all and are, in fact, identical. The small differences between the inputs that can be observed are caused by background activity on the simulated machine. The inputs for `blackscholes`, `canneal` and `freqmine` have a very high approximation error. These are workloads where linear input scaling has a direct effect on their working sets or which are very difficult to scale.

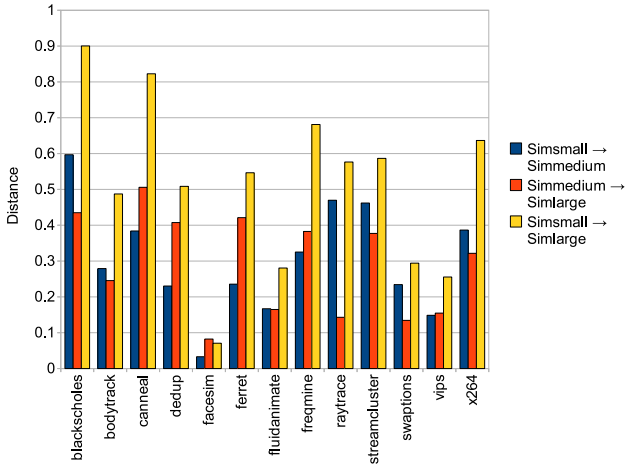


Fig. 3. Approximation error of inputs. The approximation error or dissimilarity between any two pairs of simulation inputs for the same workload is given by the distance between the corresponding points in PCA space.

V. Input Set Selection

PARSEC is one of few benchmark suites which offers multiple scaled-down input versions for each workload that were derived from a single real-world input. This requires users to select appropriate inputs for their simulation study. This section discusses how to make this decision in practice.

As mentioned in Section II-A, designing or selecting an input set is an optimization problem which has the structure of the classical multiple-choice knapsack problem. In practice, the simulation time budget T is often somewhat flexible. Thus, we can simplify the problem to the fractional knapsack problem, for which a simple greedy heuristic based on the benefit-cost ratio (BCR) leads to the optimal solution. With this approach we first determine the optimal order in which to select the inputs, then we implicitly size the simulation time budget so that there is no need to take fractional inputs. We furthermore express the input selection problem as a selection of upgrades over `simsmall` to prevent the pathological case where the greedy heuristic chooses no input at all for a given benchmark program. The cost in our case is the increase in instructions, which is a reasonable approximation of simulation time. The task of the greedy algorithms is thus to maximize error reduction relative to the increase of instructions for each input upgrade over `simsmall`.

More formally, the benefit-cost ratio of upgrading from input k to input l of benchmark i is $\mathrm{BCR}_i(k,l) = -\frac{a_{il}-a_{ik}}{t_{il}-t_{ik}}$. We use the negative value because the metric is derived from the reduction of the approximation error, which is a positive benefit. The values for the variables a_{ik} are simply the distances of the respective inputs from the corresponding reference inputs of the `simlarge` input set, which we give in Figure 3. The values for the cost t_{ik} can be measured directly by executing the workloads with the desired inputs and counting the instructions. We show all relevant BCRs in Figure 4. As can be expected, the data shows diminishing returns for upgrading to a more accurate input set: Making the step from `simsmall` to `simmedium` is always more beneficial than making the step from `simmedium` to `simlarge`.

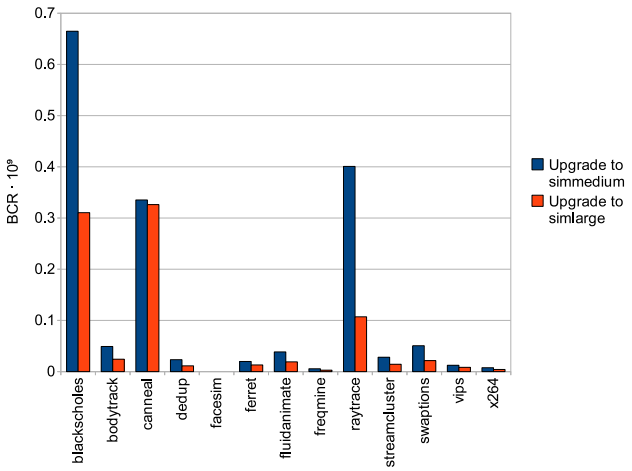


Fig. 4. Benefit-cost ratio (BCR) of using the next larger input. The chart shows the reduction in approximation error relative to the increase of cost in billion instructions. The data for `facesim` had to be omitted due to a division by zero.

As can be seen from the data, if simulation cost is considered there are three obvious candidates which are interesting for an upgrade: Using `simmedium` instead of `simsmall` is highly attractive for the workloads `blackscholes`, `canneal` and `raytrace`. The benchmarks `blackscholes` and `canneal` are furthermore interesting for another upgrade to the `simlarge` inputs, `raytrace` to a much lesser extent. The next tier of workloads with an attractive BCR is formed by `bodytrack`, `dedup`, `ferret`, `fluidanimate`, `streamcluster` and `swaptions`. These workloads also show an increased benefit-cost ratio for an upgrade from `simsmall` to `simmedium`. An interesting fact is that it is more attractive to upgrade `blackscholes`, `canneal` and `raytrace` to `simlarge` first before upgrading any other inputs to `simmedium`. This is primarily due to the fact that the inputs for these programs contain significantly fewer instructions than those of the other benchmarks. Benchmark users should furthermore verify that all selected inputs are used within their scope as defined in Section III-C. If this is not the case it might be necessary to create custom benchmark inputs, which we discuss in the next section.

VI. Customizing Input Sets

The input sets of the PARSEC release are scaled for a wide range of evaluation or simulation cases. However, for specific purposes one may want to consider designing their own input sets based on the methodology proposed in this paper. This section presents a guideline for customizing inputs for more parallelism, larger working sets or higher communication intensity.

A. More Parallelism

The amount of work units available in an input can typically be directly controlled with the linear component of the input. In almost all cases, the amount of parallelism can be increased significantly so that enough concurrency is provided by the input to allow the workload to stress CMPs with hundreds of cores. However, it is important to remember that the amount of work units contained in an input are at best only potential parallelism. It is likely that not all workloads will be able to scale well to processors with hundreds of cores. We describe some technical limitations in Section III-C.

B. Larger Working Sets

It is straightforward to create massive working sets with PARSEC. The standard input sets already provide fairly big working sets, but as mentioned in Section III, linear input scaling was used to aggressively reduce the input sizes of the simulation inputs. This can significantly reduce working set sizes if the outer-most loop has a direct impact on the amount of data that will be reused, which is especially true for benchmarks with unbounded working sets.

The general approach to obtain a larger working set with a PARSEC program is to first use complex input scaling to increase the amount of data that the program keeps in memory and then guaranteeing enough reuse of the data by linear input scaling.

Our experience shows that it is easy to underestimate the impact of increasing working set sizes on execution time. It is known that program execution time grows at least linearly with the program's memory requirements because each item in memory has to be touched at least once. However, most programs use algorithms with complexities higher than linear, which means increasing problem sizes may lead to a prohibitively long execution time. For example, increasing the working set size of an algorithm which runs in $O(N \log N)$ time and requires $O(N)$ memory by a factor of eight will result in a 24-fold increase of execution time. If the simulation of this workload took originally two weeks, it would now take nearly a full year for a single run - more than most researchers are willing to wait.

C. Higher Communication Intensity

The most common approach to increase communication intensity is to reduce the size of the work units for threads while keeping the total amount of work constant. For example, the communication intensity of `fluidanimate` can be increased by reducing the volume of a cell because communication between threads happens at the borders between cells. However, this is not always possible in a straightforward manner because the size of a work unit might be determined by the parallel algorithm or it might be hardwired in the program so that it cannot be chosen easily.

The communication intensity is limited in nearly all lock-based parallel programs. Communication among threads requires synchronization, which is already an expensive operation by itself that can quickly become a bottleneck for the achievable speedup. Programs with high communication intensity are typically limited by a combination of synchronization overhead, lock contention or load imbalance, which means that parallel programs have to be written with the goal to reduce communication to acceptable levels. The PARSEC benchmarks are no exception to this rule.

VII. Related Work

Closely related work falls into three categories [26]: *reduced inputs*, *truncated execution*, and *sampling*.

A popular method to reduce the number of simulated instructions of a benchmark is called *reduced inputs*. The `test` and `train` input sets of the SPEC CPU2006 benchmark suite are sometimes used as a reduced version of its authoritative `ref` input set. MinneSPEC [14] and SPEClite [22] are two alternative input sets for the SPEC CPU2000 suite that provide reduced inputs suitable for simulation. Our work goes beyond previous work by proposing a framework and employing a continuous scaling method to create multiple inputs suitable for performance studies with varying degrees of fidelity and simulation cost.

Truncated execution takes a single block of instructions for simulation, typically from the beginning (or close to the beginning) of the program. This method has been shown to be inaccurate [26].

Sampling is a statistical simulation method that provides an alternative to reduced inputs. It selects small subsets of an instruction stream for detailed simulation [4], [5], [20], [25]. These sampling methods choose brief instruction sequences either randomly or based on some form of behavior analysis. This can happen either offline or during simulation via fast forwarding. Statistically sampled simulation can be efficient and accurate [26], but it requires a sophisticated mechanism built into simulators.

The importance of limiting simulation cost while preserving accuracy has motivated studies that compare sampling with reduced inputs. Haskins et al. concluded that both approaches have their uses [11]. Eeckhout et al. showed that which method was superior depended on the benchmark [7]. Finally, Yi et al. concluded that sampling should generally be preferred over reduced inputs [26]. These comparisons however did not consider that the accuracy of either method is a function of the input size. Our work provides the framework to allow benchmark users to decide for themselves how much accuracy they are willing to give up for faster simulations.

Another approach is statistical simulation [8], [16], [17]. The fundamental concept of statistical simulation is to generate a new, synthetic instruction stream from a benchmark program with the same statistical properties and characteristics. These statistical properties have to be derived by first simulating a workload in sufficient detail. The statistical image obtained that way is then fed to a random instruction trace generator which drives the statistical simulator.

VIII. Conclusions

We have developed a framework to scale input sets for a benchmark suite. Our approach considers that benchmark inputs are approximations of real-world inputs that have varying degrees of fidelity and cost. By employing concepts of mathematical approximation theory and scientific modeling we have provided the methodology to allow benchmark creators and users to reason about the inherent accuracy and cost tradeoffs in a systematic way.

We have shown that the problem of choosing the best input size for a benchmark can be solved in an optimal way by expressing it as the multiple-choice knapsack optimization problem. The inputs can then be selected by standard algorithms so that benchmarking accuracy is maximized for a given time budget. For practical situations the problem can be further simplified so that it can be solved optimally with a simple greedy heuristic.

We have quantified the approximation errors of multiple scaled input sets of the PARSEC benchmark suite and have suggested a sequence of input upgrades for users to achieve higher simulation accuracies for their simulation time budgets. We have also analyzed the constraints of the PARSEC simulation inputs to provide users with the scope of architectures for which the inputs exhibit reasonably accurate behavior.

We have also given guidelines for users to create their own input sets for PARSEC benchmark programs with a scope more fitting for their specific simulation purpose. The proposed scaling model is used to categorize the various input parameters of the PARSEC benchmarks, which gives users a better understanding of the input creation process and its impact on program behavior.

References

[1] M. P. Bailey and W. G. Kemple. The Scientific Method of Choosing Model Fidelity. In *Proceedings of the 24th Conference on Winter Simulation*, pages 791–797, New York, NY, USA, 1992. ACM.

[2] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of the 2008 International Symposium on Workload Characterization*, September 2008.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[4] P. D. Bryan, M. C. Rosier, and T. M. Conte. Reverse State Reconstruction for Sampled Microarchitectural Simulation. *International Symmposium on Performance Analysis of Systems and Software*, pages 190–199, 2007.

[5] T. Conte, M. Ann, H. Kishore, and N. Menezes. Reducing State Loss For Effective Trace Sampling of Superscalar Processors. In *Proceedings of the 1996 International Conference on Computer Design* , pages 468–477, 1996.

[6] K. Dudzinski and S. Walukiewicz. Exact Methods for the Knapsack Problem and its Generalizations. *European Journal of Operations Research*, 28(1):3–21, 1987.

[7] L. Eeckhout, A. Georges, and K. D. Bosschere. Selecting a Reduced but Representative Workload. In *Middleware Benchmarking: Approaches, Results, Experiences. OOSPLA workshop*, 2003.

[8] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. D. Bosschere. Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox. *IEEE Micro*, 23:26–38, 2003.

[9] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003.

[10] R. Giladi and N. Ahituv. SPEC as a Performance Evaluation Measure. *Computer*, 28(8):33–42, 1995.

[11] J. Haskins, K. Skadron, A. KleinOsowski, and D. J. Lilja. Techniques for Accurate, Accelerated Processor Simulation: Analysis of Reduced Inputs and Sampling. Technical report, Charlottesville, VA, USA, 2002.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[13] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring Benchmark Similarity Using Inherent Program Characteristics. *IEEE Transactions on Computers*, 28(8):33–42, 1995.

[14] A. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, 2002.

[15] R. M. Nauss. The 0-1 Knapsack Problem with Multiple-Choice Constraints. *European Journal of Operations Research*, 2(2):125–131, 1978.

[16] S. Nussbaum and J. E. Smith. Modeling Superscalar Processors via Statistical Simulation. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.

[17] S. Nussbaum and J. E. Smith. Statistical Simulation of Symmetric Multiprocessor Systems. In *Proceedings of the 35th Annual Simulation Symposium*, page 89, Washington, DC, USA, 2002. IEEE Computer Society.

[18] A. Phansalkar, A. Joshi, and L. K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 412–423, New York, NY, USA, 2007. ACM.

[19] D. Pisinger. A Minimal Algorithm for the Multiple-Choice Knapsack Problem. *European Journal of Operational Research*, 83:394–410, 1994.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large-Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[21] P. Sinha and A. A. Zoltners. The Multiple-Choice Knapsack Problem. *Operations Research*, 27(3):503–515, 1979.

[22] R. Todi. SPEClite: Using Representative Samples to Reduce SPEC CPU2000 Workload. In *Proceedings of the 2001 IEEE International Workshop of Workload Characterization*, pages 15–23, Washington, DC, USA, 2001. IEEE Computer Society.

[23] Vandierendonck, H. and De Bosschere, K. Many Benchmarks Stress the Same Bottlenecks. In *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 57–64, 2 2004.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[25] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–97, 2003.

[26] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 266–277, Washington, DC, USA, 2005. IEEE Computer Society.

Exploiting Approximate Value Locality for Data Synchronization on Multi-core processors

Jaswanth Sreeram
College of Computing
Georgia Institute of Technology
jaswanth@gatech.edu

Santosh Pande
College of Computing
Georgia Institute of Technology
santosh@cc.gatech.edu

***Abstract*—This paper shows that for a variety of parallel "soft computing" programs that use optimistic synchronization, the approximate nature of the values produced during execution can be exploited to improve performance significantly. Specifically, through mechanisms for imprecise sharing of values between threads, the amount of contention in these programs can be reduced thereby avoiding expensive aborts and improving parallel performance while keeping the results produced by the program within the bounds of an acceptable approximation. This is made possible due to our observation that for many such programs, a large fraction of the values produced during execution exhibit a substantial amount of *value locality*.**

We describe how this locality can be exploited using extensions to C/C++ language types that allow specification of limits on the precision and accuracy required and a novel value-aware conflict detection scheme that minimizes the number of conflicts while respecting these limits. Our experiments indicate that for the programs studied substantial speedups can be achieved - upto 5.7x over the original program for the same number of threads. We also present experimental evidence that for the programs studied, the amount of error introduced often grows relatively slowly.

I. Introduction

There is a large class of real-world programs termed *Soft Computing applications* [13] which are characterized by several unique properties.

- **Approximate nature of results.** These applications all produce an approximation of the actual results rather than their actual values. This may be because of several reasons. One common reason is that the physical or mathematical model expressed in the program requires some approximation to be computable in a reasonable amount of time. Other programs such as simulation applications mimic continuous processes but in a discrete-time fashion and this introduces some error in the result.
- **User-defined correctness.** In some cases, the application programmer can choose to consciously sacrifice accuracy of the results in order for the program to meet some execution characteristics such as soft real-time deadlines. He or she may be able to control parameters that directly determine the amount of error in the results produced. Examples of such parameters include thresholds in approximations, the granularity of ticks in time-stepped simulations, cutoff distances and radii in physical simulations etc.
- **Tolerance for Imprecision and Uncertainty.** Soft computing applications to some extent are tolerant of imprecision in inputs and some program values. Many such applications are designed to work with input streams and program values which are inherently noisy, imprecise or unreliable. Examples of such programs include pattern recognition systems, object-tracking systems and other machine learning applications.

Several researchers have shown that for many such soft computing programs, it is possible to design optimizations that exploit these properties to improve performance by sacrificing the accuracy, precision or some other aspect of intermediate computations and of the result produced [17], [24]. In [26] the authors propose an FPU and architecture design that uses dynamic precision reduction for lower energy and area requirements. In this paper we study the phenomenon of *store value locality* and its application to reducing synchronization conflicts in programs that use optimistic concurrency control such as hardware or software transactional memory systems.

A. Value-aware Synchronization

In a multithreaded program on a shared memory machine, shared variables are used to communicate values between different threads. This communication is synchronized using explicit constructs such as locks and mutexes or in the case of an optimistic synchronization system such as a hardware or software transactional memory system (H/STM) it is guaranteed by the runtime provided the programmer follows the constraints on specifying atomic sections correctly. For two concurrent threads, a write to a shared variable in one thread signifies production of a new value that may be consumed in the other thread. This production and consumption of values are usually synchronized precisely. However in many soft computing applications, the program may be tolerant of some level of imprecision in this synchronization. In the most common case, the consumer of a value from a shared variable may be able to proceed with its computation without receiving the newest value produced into that variable provided that the newest value produced is not too different from the old value that it read. That is, if consecutive updates made to the shared state are relatively small, then the consumer may be able to proceed with the older state without waiting for the newest value, as happens in normal (precise) synchronization. In the following sections we show that for many programs a large fraction of dynamic writes update shared variable in this manner. We also show that this property combined with the properties of soft computing applications described previously allow us to reduce synchronization overheads and improve parallel execution performance.

The three major contributions of our work and the organization of the rest of the paper are outlined below:

- We describe the phenomenon of *Approximate Store Value Locality* and show experimental evidence that establishes the existence of this phenomenon in many programs (Section II).
- Given a similarity threshold, we propose a mechanism for detecting Approximate Store Value Locality efficiently in a program that uses optimistic synchronization (Section VI-A)
- We describe a technique to exploit this locality phenomenon in reducing the number of conflicts in several soft computing applications which are tolerant to imprecise sharing of data

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

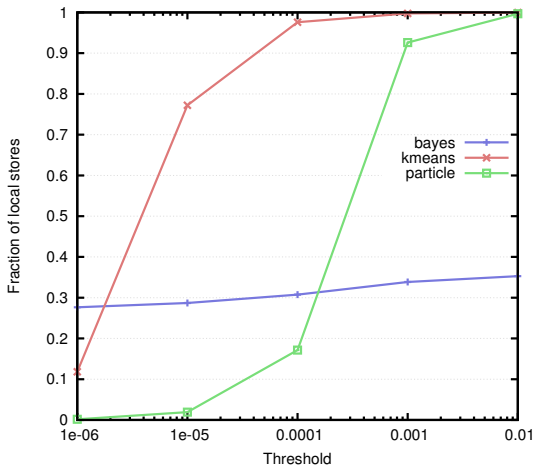


Fig. 1: Approximate Shared Value Similarity in Critical Sections

between threads (Section VI-B) and present an experimental evaluation of performance and accuracy (Section VII).

II. Approximate Store Value Locality

The phenomenon of *Store Value Locality* (SVL) in programs has been reported and studied widely in literature [5]. Briefly, a program is said to exhibit store (or shared) value locality when many write operations in the program write values that are either trivially predictable or exactly match the values already at the memory address being written. In this section, we show that a related but different property of *Approximate Store Value Locality* is also prevalent for many programs. This term describes the phenomenon where many writes write values that are *approximately local* to the values already at the memory address being written. We define *"approximate locality"* of two values v_0 and v_1 to be as follows:
" Two values v_0 and v_1 are approximately local for a small threshold τ if $\mid v_0 - v_1 \mid < \tau$"

Therefore if a store instruction is about to write v_1 and the value v_0 is already present in memory at that address and the above condition is met, we say that the instruction exhibits *Approximate Store Value Locality* (ASVL) for the threshold τ and we call this store an *approximately-local store*. Whether a particular segment of code exhibits ASVL depends on the value of τ and the values themselves.

A. Approximate Value Locality in Critical Sections

In many real world applications, many of the values produced into shared variables in critical sections, undergo transformations that change them very little in relative terms. To test this hypothesis, we collected statistics on approximately value locality for the programs shown in Figure 1. Specifically, we measured what percentage of stores to shared floating point variables inside transactional code committed values that were approximately similar to the values already present. The results are shown in Figure 1. In this graph the X-axis corresponds to the relative similarity between values written by stores to the same shared memory location. The Y-axis shows the percentage of total number of dynamic stores operations inside critical sections, that are exhibited this value similarity. We see that for all the programs shown, a substantial fraction of dynamic writes inside transactional code were *approximately local stores*. In these programs there are a lot of single or double precision floats and indeed in many cases most of the computation inside transactional code is performed on these floats - the number of approximately local stores that wrote integers was insignificantly low in all cases. These statistics tell us that a significant portion of shared values produced inside critical sections are arithmetically similar (the overall similarity being a function of the threshold). Since shared variables are typically used for communicating state or updates to state between threads, a related observation we can make is that for these programs,
"A significant portion of the values or updates being exchanged between the threads are relatively close to each other in magnitude"

In [5], the authors cite several reasons for the existence of store locality in real world programs. In addition to those factors, there are a few other empirical reasons that explain the ASVL phenomenon

- **Similarity in input data**: Many real-world input data sets contain a substantial number of input values that are similar.
- **Iterative refinement**: Many critical sections occur inside loops where the results computed in the loop body are synchronized with the global state at the end of each iteration. If the results computed are similar or approximately similar for two consecutive iterations (i.e., each thread, modifies global state by a relatively small magnitude), then the store in the critical section that updates global state will often exhibit the ASVL property.
- **Finite Precision**: All real hardware has finite precision. Therefore knowing whether a *silent store* has occurred is itself an approximate endeavor if the store was writing a floating point value. Hence, for many programs which make heavy use of floating point numbers Store Value Locality manifests as Approximate Store Value Locality.

Most optimistic data synchronization mechanisms like transactional memories operate on meta-data such as versions and are oblivious to the actual values being shared between threads. Therefore systems with TMs, speculative lock-elision mechanisms etc., are unable to detect or exploit the approximate shared value similarity phenomenon. In Sections III VI we develop techniques to do both. While these techniques are discussed in the context of a TM system the broad principles apply to other optimistic synchronization systems as well.

III. Strong False-conflicts

In a transactional memory system, two concurrent transactions are said to conflict if both of them access the same shared variable and at least one of them performs a write operation on that variable. When such a conflict occurs, at least one of the transactions (usually the reader) is aborted. For example, consider the concurrent conflicting transactions T_1 and T_2 with the schedules below:
T_1(`start`); T_1(`write` v_1 in x); T_1(`commit`)
T_2(`start`); T_2(v_0 = `read` x); T_1(`commit`)

The TM system detects this conflict by determining whether the value read by T_2 could have been modified by T_1. Most TM systems typically use meta data such as version numbers with or without global clocks.

In TMs that use only version numbering, each shared variable or region of memory that can be accessed transactionally is associated with a version number. During a transactional read/write of this variable, this version number is cached by that transaction. A committing transaction increments the version

numbers of all the variables that it is writing to (T_1 would increment the version number for x when it commits). During the commit phase, the version number cached for each variable read/written is compared to the latest committed version number for that variable. If the version number is the same, then there could not have been any writes to that variable since this transaction started. If the version number is different, then some other transaction must have written to this variable, and incremented its version number and a conflict is detected. Several other TM systems such as TL2 [11] additionally use the notion of global version-clocks, to order transaction start, read, write and commit events. In such systems, there is a global shared clock whose value (g) each new transaction reads when it starts. For each variable that can be accessed in a transaction there is a versioned write-lock (l). Each transaction also creates a local copy (wv) of the "write-version" by incrementing and fetching g. When a transaction wants to commit, it first iterates through its read and write sets to check if the corresponding l for that variable is less than g. If so, it is safe to commit. During the commit phase, the transaction iterates through its write set and for each variable therein, stores its new value from the write set and updates its versioned lock l to wv. In both types of systems described above and in general for most TM systems, a conflict for a shared variable is detected by comparing some local meta data for that variable with some global meta data. This method of detecting conflicts can result in pseudo-conflicts if the transaction commits the same or similar value as was present originally before the transaction started. Thus, if the concurrent transactions T_1(reader) and T_2(writer) have been found to conflict and T_2 commits the same value as existed when T_1 read it (i.e., the committing store operation in T_2 was a *silent store*), then we call this conflict a *strong false-conflict*. Two distinct transaction schedules where this occurs are shown below:

T_1(`start`); T_2(`start`); $T_1(v_0 =$ `read` $x)$; T_2(`write` v_0 in x); T_2(`commit`); T_1(`commit`)

T_1(`start`); $T_1(v_0 =$ `read` $x)$; T_2(`start`); T_2(`write` v_0 in x); T_2(`commit`); T_1(`commit`)

We call these conflicts "strong false-conflicts" because ignoring them during the conflict resolution phase would not affect the correctness of the program. Redundant store operations such as the one in T_2 above can be eliminated by traditional compiler optimizations if the compiler is able to assert that v_0 is already the value at address x. However, this ability is restricted by procedure calls, indirect branches and other conditions in which the compiler cannot guarantee this condition is met.

IV. Weak False-conflicts

The definition of "false conflict" requires us to define clearly what "equivalence" is. Determining equivalence is straightforward for data types such as integers and fixed point values, but is not well-defined for single or double precision floating point values since such values are represented with quantities with finite precision on any hardware. For floating point variables, we can only assert whether the two differ by at most some given value. This approximate equality is routinely used to compare floating point values in programs where the threshold for two floats to be considered equal, is supplied by the programmer. We call two floating point values to be "similar for threshold τ" if the difference between these values is smaller than τ.

We now extend the notion of false conflicts to include those caused by writing a value that was within some threshold of the original value that existed at that memory address. Therefore *for two concurrent transactions T_1 and T_2 accessing a shared variable x, if the value v_0 read by T_1 is overwritten with v_1 by transaction T_2 before T_1 commits and v_0 and v_1 are approximately-local for a threshold τ, then T_1 and T_2 are said to have a weak false conflict for τ*

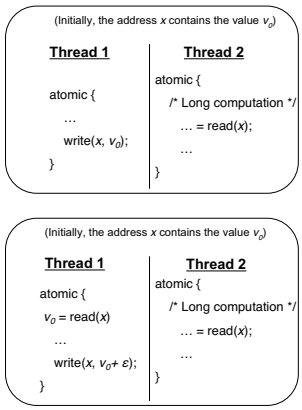


Fig. 2: Example of two threads with Strong and Weak False-conflicts

That is, for the following transaction schedule:

T_1(`start`); $T_1(v_0 =$ `read` $x)$; T_2(`start`); T_2(`write` v_1 in x); T_2(`commit`); T_1(`commit`)

if $\mid v_0 - v_1 \mid < \tau$ then this schedule has a weak false conflict. This notion of similarity for a given threshold is well defined for native data types such as signed and unsigned integers and floating point values and fixed point values.

These false-conflicts arise because most widely used conflict detection mechanisms do not take into account the actual values being read and written and instead only use versions or such meta data. On the other hand, threads and the atomic units inside them care about values and not version numbers. Specifically, a reader of a shared variable does not usually care whether a particular variable has a different version number provided it has the same (or in some cases, approximately similar) value. This would suggest that employing a conflict detection and resolution mechanism that had the ability to inspect the actual values of shared variables would improve concurrency at the cost of physical serializability.

V. Specifying Imprecise Sharing

Many real-world programs use thresholds for program quantities that results in an approximate final answer (often to improve execution time). Examples of such thresholds are cutoff radii in particle or molecular dynamics simulations, thresholds for scores in pattern matching and object recognition programs, timestamp windows for event processing in discrete event simulation systems and so on. In other programs, the programmer implicitly specifies this by having controlled, deliberately lazy updates to global shared data (for example in the Bayesian network simulation in [6]) and deliberate uses of stale, unsynchronized global state (many real-time particle systems such as in video games), among others. In all of these examples, there is some implicit or explicit specification by the programmer regarding what set of conditions can lead to an approximate but acceptable final answer. In our system, this specification is based on assertions about the *similarity* between values. Specifically, the programmer can use the threshold τ for specifying these assertions, knowing that given a program value a and its threshold τ_a, any value within τ_a distance of a is treated as semantically equivalent to a. All the

implicit and explicit specifications for program approximations described above can be captured using this form.

A. Choice of Comparison Functions

In the previous section we defined the notion of weak false-conflicts for a particular threshold τ due to *approximate locality*. Determining approximate locality requires a robust and well-defined *Thresholded-Comparison* operation. In the simplest case described above this operation was simply an absolute comparison function and the τ was also absolute. However this is inadequate since expressing absolute thresholds for changes in program values requires the programmer to be aware of the magnitudes of initial, final and intermediate program values. Instead our system uses the following comparison operations that use relative thresholds.

- $RelativeError(a, b, \tau_r)$: This operation determines if the *relative error* between values a and b is lower than the τ_r which is simply a float that describes the maximum permissible relative error. One well known problem with this operation is that it fails for numbers very close to zero. The positive float (double) closest to zero and the negative float (double) closest to zero are very close to each other but this function will determine them to be very far apart. However for other values this operation is fairly robust and intuitive to use.
- $MaxValues(a, b, \tau_u)$: This operation determines if the number of $representable$ values between a and b is less than τ_u which is an integer. Therefore if this operation returns "true" for two floats a and b for $\tau_u = 1000$, this means that there are at most 1000 representable floats between a and b. This operation is more robust than $RelativeError$ but it requires reasoning about thresholds in terms of number of representable values between two program values.

B. Thresholded Types

The comparison functions described above are sound for integers, single and double precision floats. We define a set of augmented types that extend the native types with a threshold and a comparison function. These types are shown in Figure 3. Here `REL` and `MV` refer to the $RelativeError$ and $MaxValues$ functions respectively.

Scope: While avoiding weak false-conflicts may improve performance in many cases improper use of the thresholded types can inject error into the computation that renders the outputs meaningless or worse results in catastrophic failure of the program. Below, we list some important considerations in using thresholds for shared data:

1) Smoothly changing values: The shared value to which a threshold is being applied should change smoothly relative to the threshold. Otherwise the consumer thread may observe values that change drastically.
2) Flag variables and predicates: Flag variables and predicates should not be thresholded as this will result in control flow being drastically changed.
3) Pointers: While the notion of a thresholded pointer may be useful in some cases, pointers should not be thresholded. This is because calls to functions such as `realloc` may leave the value in the pointer intact, but may change the attributes of the data or buffer being pointed to. Only native signed/unsigned integers, single/double precision floats should be thresholded.
4) Invariants: Programmatic or algorithmic invariants can be very useful in determining or controlling the amount of error introduced by using thresholded types. For example in a physical simulation for a closed particle system (discussed in detail in Section VII) the total energy in the system is a physical invariant (due to the first law of thermodynamics). Therefore the amount of tolerable error can be specified as a function of deviation allowed from this invariant and the thresholds can be determined accordingly.
5) Knowledge of program behavior: Finally, the value of the threshold for a particular variable must be based on the programmer's knowledge of the system being modeled and of the magnitudes of the quantities being computed. Just like STM features such as *partial commits*, *early release* and other mechanisms that affect the serializability and/or the correctness of the program, these imprecise synchronization techniques should only be used by expert programmers in situations when the implications are clear.

```
// Types using the "REL" function
typedef float(REL, 0.0001)
RELThreshFloat
typedef double(REL, 0.0000001)
RELThreshDouble
RELThreshFloat x;
RELThreshDouble z;

// Types using the "MV" function
typedef float(MV, 1000)
MVThreshFloat
typedef double(MV, 1000000)
MVThreshDouble
MVThreshFloat a;
MVThreshDouble c;
```

Fig. 3: Extensions to native types for specifying thresholds and comparison functions

These thresholds are bound to the variables over the scope of the transaction and they will be used during the conflict detection phase described below.

VI. Avoiding Strong and Weak False-conflicts

We defined a false conflict as one resulting from a silent or an approximately silent store operation in a thread that overwrote a value in memory with the same value or with a value that differed from it by a small τ. Certainly, in the case where τ is exactly equal to 0, (i.e., the store operation wrote the exact same value as the one already existing at that address), the conflict is not a real conflict since although ignoring it would affect the physical serializability of the transaction schedule, it would not affect the semantics of either transaction or those of the values produced therein. Therefore eliminating these conflicts would reduce transaction abort rate and therefore overall transactional throughput. Even for non-zero values of τ, many programs (or transactions in programs) can tolerate this approximate sharing of values and hence we would like to avoid weak false conflicts for a given τ. To do this we need efficient methods for two tasks - detecting approximate store value locality and avoiding conflicts that result from the occurrence of this locality.

A. Detecting Approximately-Local Stores

There has been substantial amount of work on detecting silent stores efficiently [5], [19] and a majority of them are based on program profiling and/or special purpose hardware

for tracking stores. These works are discussed in Section VIII. In our system however, instead of tracking silent stores, we want to detect Approximately Store Value Locality, i.e., store instructions that write a value that is within some small τ of the value already present at the address being written to. Moreover we would like to do this without the aid of profiling or special hardware. Fortunately, the use of optimistic synchronization (as in a TM system) for critical sections gives us a channel to monitor store values dynamically with little additional cost.

We will describe the detection technique in the context of a TM system like TL2 [11]. In TL2, there is a global version-clock variable g that is read and written by each writing transaction and is read by each read-only transaction. In addition each transacted memory location has a versioned write lock l that consists of a 1-bit $write\text{-}lock$ predicate that indicates whether the lock is currently held by some thread and a *lock version* field that indicates the version number of the variable at that instant. Also, each transaction also has a table containing mappings from shared variables accessed in that transaction to their respective τ values that were specified by the programmer in the original program. At commit time a transaction attempts to acquire write locks for each of the elements in its write-set. If it is successful it will then perform an atomic increment-and-fetch operation on the value of g and record the returned value in a *local write-version number* variable wv. This value of wv is essentially the version number that this transaction will give to all the variables in its write-set after it has written to them. Then validation of the read-set is performed. If this is successful, a write-back is performed where for each variable a in the write-set, its new value buffered in the write-set is written to memory. Just before this write happens, we can determine whether the new value that is about to be written and the old value at that address are *approximately local* for a threshold value τ_a. If so, we can simply mark this variable in the write-set as having been written to by a store exhibiting approximate store value locality. This step is shown in Algorithm 1 which is implemented in the commit protocol for the TM.

The cost of both the *RelativeError* and *MaxValues* comparison functions *is independent of the magnitude of the values being compared and is of the order of tens of instructions per comparison*. This is important for transactions that have large read/write sets and which therefore may invoke these functions frequently.

Algorithm 1 Detecting *Approximate Store Value Locality*

```
Require: Transaction T
  // Transaction writeback
  for all e ∈ WriteSet do
    if computeSimilarity(e.newVal, e.oldVal) == true then
      e.similarityflag = 1
    end if
  end for
  // Drop Locks
  for all e ∈ WriteSet do
    if e.similarityflag ≠ 1 then
      e.version = T.wv
    else
      e.version = T.rdv
    end if
  end for
```

B. Avoiding Conflicts due to Approximately-Local Stores

After a committing transaction has finished writing back the new values it has produced, it releases the write locks it holds and then clears the $write - lock$ bit for each variable in its write-set. The process of releasing a write lock for an element in the transaction's write set essentially consists of setting the *lock version* for that memory location to the local write version wv recorded in the transaction when it started its attempt to commit. This signifies a new value as having been produced and committed in the system.

In the detection phase described in Section VI-A above, we identified and marked all variables in a committing transaction which were written to by an approximately-local store. Therefore while releasing locks for each variable in the write set in the current phase, we check to see if this variable was marked. If it was not, then the lock is released normally. If it was marked, then we bypass the updating the *lock version* for that variable to wv. Hence this variable contains the same version number as it did before this transaction acquired a lock on it. This is all the committing transaction needs to do.

A transactional read of a variable proceeds as follows. Before the load for the variable is executed, two other load instructions are first executed. The first one checks if the 1-bit write-lock is free. If it is set, then some other transaction is currently writing to this location and the transaction fails. If it is not set, then the *lock version* field *wv* is checked to make sure that it is lower than the transaction's read version *rv*. If it is greater than *rv*, then some other transaction has committed to it after the current transaction started. In the detection phase above, the *wv* field is not updated if the committing store is found to be *approximately local*. To see why this technique reduces conflicts, consider the example of a reader transaction T_2 in Thread 2 and a concurrent writer transaction T_1 in Thread 1 both accessing a variable x as shown in Figure 2. Let us assume T_2 started first. It first read the *global version clock* $g(=0)$ into the thread local read-version variable rv_2. It then reads the value in x then proceeds to perform some computation. Sometime after that transaction T_1 starts and records the value $g(=0)$ in its local read-version variable rv_1. It then executes an *approximately-local* store to x (which updates the value for x in its write-set). The transaction T_1 then attempts to commit. It updates g from 0 to 1 and sets its own wv_1 to 0 and then attempts to write its write-set to memory.

During this step, the ASVL detection mechanism described above is invoked and it marks the x in T_1's write-set as approximately local. The new value $(v_0 + \epsilon)$ is then written to memory. Then T_1 attempts to release the write-lock on x and here the conflict avoidance mechanism described above is invoked. T_1 checks if the x in its write set is marked. Since it is marked, the *lock version* for x is *not* updated to wv_1 and is left unchanged at 0. Then if T_2 tries to commit, it checks to see if the *lock version* $\leq rv_2$. This will turn out to be true, since *lock version* = rv_2 = 0. Now T_2 can proceed to commit successfully.

This example describes one scenario and several others are possible with different orderings of transaction starts, reads, writes and commits.

VII. Experimental evaluation

A. Experimental Setup

We implemented the false-conflict detection and avoidance techniques described above in the TL2 STM system. In this section we present results of our experimental evaluation of the techniques along two dimensions. Firstly we evaluated the effectiveness of our techniques in reducing the number of

false conflicts and the aborts caused due to them. Secondly, we studied the amount and nature of error introduced in the program and the trade-off between accuracy and performance. We present these results below using case studies of three well-known parallel programs that can be characterized as *soft computing applications* according to the criteria outlined in I. The programs are `bayes` and `kmeans` from STAMP and `particle`. All programs were compiled with gcc-4.2 and executed on an machine with an Intel Quad Core processor with 4 hyperthreaded cores, each with an 8K L1 cache and 128K L2 cache running Ubuntu Linux. All running times were gathered using the gettimeofday() call. To minimize the interference due to system thread scheduling each thread was statically bound to a specific core. In all the experiments discussed below thresholds were applied only for single and double precision float types.

B. Case Studies

1) `Bayes`: A Bayesian network [6] is a way of representing probability distributions for a set of variables in a concise and comprehensible graphical manner. A Bayesian network is represented as a directed acyclic graph where each node represents a variable and each edge represents a conditional dependence. By recording the conditional independences among variables (the lack of an edge between two variables implies conditional independence) a Bayesian network is able to compactly represent all of the probability distributions.

Bayesian networks have a variety of applications and are used for modeling knowledge in domains such as medical systems, image processing, and decision support systems. For example a Bayesian network can be used to calculate the probability of a patient having a specific disease given the absence or presence of certain symptoms.

Algorithm 2 `Bayes`

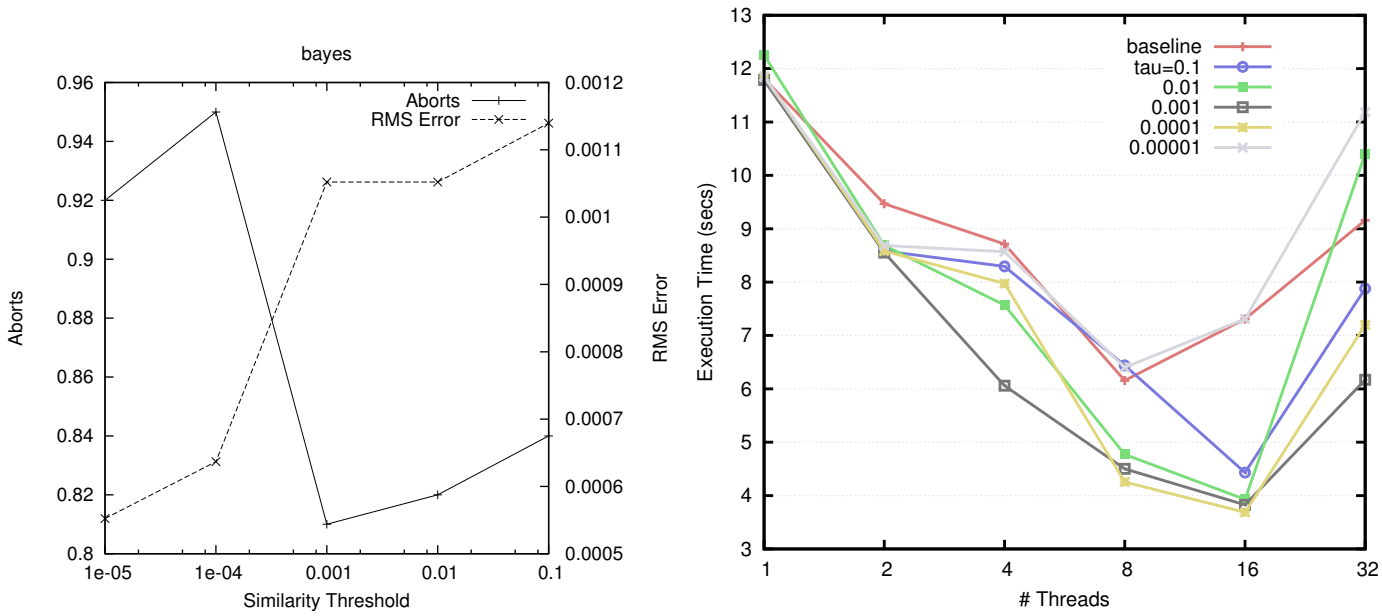
```
while (task = popTask()) ≠ NULL do
  if task.op→isInsert() then
    toID = task.toID
    newbll = computeLocalbll(toID)
    atomic {
    t = tm_read(localbll[toID])
    d += t - newbll
    tm_write(localbll[toID], n)
    } endatomic
  end if
  atomic {
  oldbll = tm_read(g_bll)
  newbll = oldbll + d
  tm_write(g_bll, newbll)
  } endatomic
  findAndInsertNextTask()
end while
```

This application implements an algorithm for learning Bayesian networks from observed data. The algorithm implements a hill-climbing strategy that uses both local and global search. The broad outline of the algorithm is shown in Algorithm 2. The network starts out with no dependencies between variables and the algorithm incrementally learns dependencies by by analyzing the observed data. On each iteration each thread is given a variable to analyze and as more dependencies are added to the network connected subgraphs of dependent variables are formed. A transaction is used to protect the calculation and addition of a new dependency, as the result depends on the extent of the subgraph that contains the variable being analyzed.

a) ***Computation of total base log likelihood****:* The global base log likelihood (*g_bll* in Algorithm 2) is computed by computing the local base log likelihood for each variable, accumulating it, and finally atomically incrementing the current global log likelihood with this accumulated value. The application already implements an approximation wherein local log likelihoods are not communicated across threads to improve performance. We extend this by specifying a threshold τ for which the store of the global log likelihood can be considered *approximately similar* if the current global log likelihood is within that threshold. The τ is relative and so we use the $RelativeError$ operation.

We show the amount of error and number of aborts for several threshold values in Figure 4a. The X-axis represents the thresholds used on a logscale. The left Y-axis shows the abort rate and the right Y-axis shows the corresponding amount of error in the result. The `bayes` program computes a learned score that it has learned from the observed data in addition to an actual score. We computed the amount of error in the learn scores produced by the program relative to the baseline and normalized this difference with the difference between lent and actual scores in the baseline case (which is the original program running with 4 threads). We see from the Figure 1 that a significant portion of dynamic stores are approximately similar for `bayes`. From Figure 4a we see that the number of aborts is reduced by almost 19% for a threshold of 0.001 producing a final error of roughly 5.3E-4. The calculation of new dependencies take up most of the execution time in this application causing it to spend almost all its execution time in long transactions that have large read and write sets. This program also has a high amount of contention as the subgraphs change frequently. Therefore by alleviating some of this contention through imprecise synchronization we are able to reduce the number of aborts which would in turn lead to improved execution time (since long running transactions implies a high penalty for aborting them). **One important property of this program is that the number of aborts and execution time depend on the order in which edges are inserted into the graph**. Therefore we expect the speedups as shown in Figure 4b to not be as smooth (see [6]).

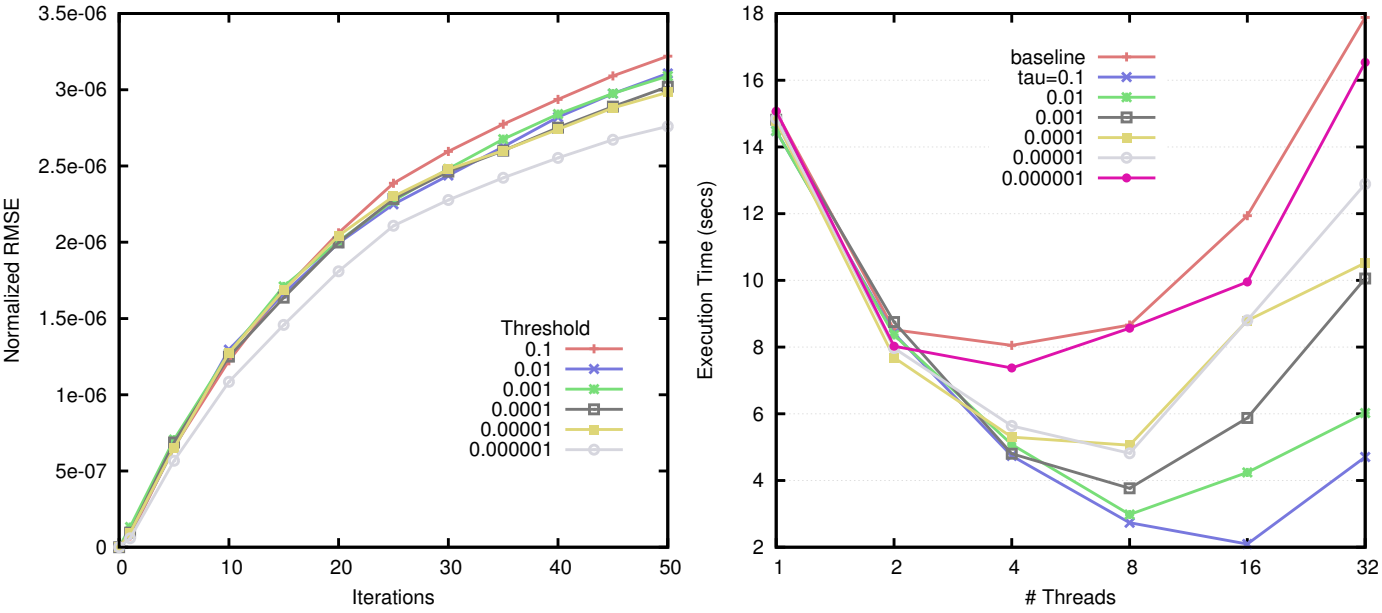
2) `Kmeans`: The K-means algorithm [6] is a partition-based method to group objects in an N-dimensional space into K Clusters. It is commonly used to partition data items into related subsets, a common operation in many data mining applications. K-means represents a cluster by the mean value of all objects contained in it. The `kmeans` program in STAMP implements the K-means algorithm that is shown in Algorithm 3. Given the user-provided parameter k the initial k cluster centers are randomly selected from the database. Then each thread in the program is given a partition of the objects which it processes iteratively. Processing an object essentially consists of assigning the object to its nearest cluster center according to a similarity function. The Euclidean distance between the object and the cluster center is used as a similarity function. Once all objects in a partition have been processed new cluster centers are found by finding the mean of all the objects in each cluster. This process is repeated until two consecutive iterations generate similar cluster assignments i.e., there is no further reassignment of objects from one cluster center to another. The TM version of K-means adds a transaction to protect the update of the cluster center that occurs during each iteration. The amount of contention among threads depends on the value of K. When updating the cluster centers the size of the transaction is



(a) Error Vs. Aborts with REL threshold

(b) Execution time with REL threshold

Fig. 4: bayes



(a) Growth of error with REL threshold

(b) Execution time with REL threshold

Fig. 5: kmeans

proportional to the dimensionality of the space. Thus, the sizes of the transactions in kmeans are relatively small and so are its read and write sets. A conflict typically happens when a thread reads a cluster center for computing the distance from an object and another thread writes a new value for that cluster center.

a) ***Computation of Cluster Centers.****:* The cluster centers are computed by summing the objects within each cluster. These centers are computed and stored atomically in a transaction as shown in Algorithm 3. In the next iteration the distance of each point in a partition from all the cluster centers is computed. For a random distribution of initial cluster centers and objects, the relative amount of change in the position of the cluster centers is quite small over successive iterations. Therefore, we can apply an approximate locality threshold τ to the shared variables holding the positions of the cluster centers. Consider a thread A that has read the position of a particular cluster center in order to compute its distance from objects in A's partition. Now the thread B that owns this cluster center computes a new

Algorithm 3 `Kmeans`

```
while delta > 0 do
  delta = 0
  for all Object "i" do
    atomic {
    cc = findNearestClusterCenter(i)
    } endatomic
    if membership[i] ≠ cc then
      membership[i] = cc
      delta += 1
    end if
  end for
  for all Cluster "c" do
    atomic {
    c→center = computeNewCenter(c)
    } endatomic
  end for
end while
```

cluster center which may be less than τ away from the current cluster center that A has read. Therefore the store executed by B is an *approximately local* store and would be marked as such. When thread A finishes computing the distances of each of its objects from the old cluster center these distances may be inconsistent. However if the relative magnitude of this inconsistency is small A can go ahead with the next step of reassigning objects instead of aborting and restarting.

We see from Figure 1 that a substantial portion of the values computed for cluster centers are approximately local. Figure 5 shows the error and performance characteristics for this program with a relative threshold and the `REL` comparision operation. From Figure 5a we notice that for a relative threshold of roughly 0.0001 an error of about 3E-6 was introduced. To compute the error introduced in the computation we calculate the root mean square error *RMSE* across all dimensions of all the points in the space. This RMSE is then normalized with the size of the space (which is the magnitude of the distance between the farthest points). The normalized RMSE is remains relatively small and grows smoothly across iterations during the execution of the program as shown in the Figure.

In this program, the amount of contention among threads depends on the value of K, with larger values resulting in less conflicts as it is less likely that two threads are concurrently operating on the same cluster center. However, even with large values of K, simply increasing the data set size (the number of points) increases contention among threads [6] and this effect was very apparent in our experiments. Therefore even though the algorithm was designed to be a low-contention one, the actual contention was quite high and consequently our relaxation technique produces significant improvement in transaction success rate. **Furthermore our experiments showed that the errors in final output for this program are comparable (around 30-50% more than) to the RMSE in the outputs between several different runs of the *baseline* versions themselves**. The plot in Figure 5b shows the speedup in execution time using the `REL` comparison operation with τ ranging from 1E-4 to 1E-6. A maximum speedup of roughly 5.7x is achieved for this program with 16 threads.

3) `particle`: Particle system simulations model the evolution of complex structure and motion of particles in a given system from a relatively small set of rules [23]. Such systems have been used in diverse scenarios ranging from stochastic modeling, molecular physics to real-time simulation and computer gaming. Particle systems have also been widely studied in the context of parallelization.

The specific particle system we describe here is similar to the one discussed in [23]. It consists of a number of particles distributed among a number of threads with each thread processing a distinct block of particles. Each particle has a position vector, a velocity vector and a mass associated with it. Each of the particles experiences two forces - a constant force (such as gravity) and also the gravitational force between pairs of particles. The system evolves in time-steps and, at each time-step, the movement of the particles due to these forces is computed using numerical integration methods. The outline of the algorithm is shown in Algorithm 4. The algorithm uses Euler integration to calculate the values of the position and velocity attributes of a particle p using the following equation

$$f_p(t+dt) = f_p(t) + dt * f_p^{'}(t)$$

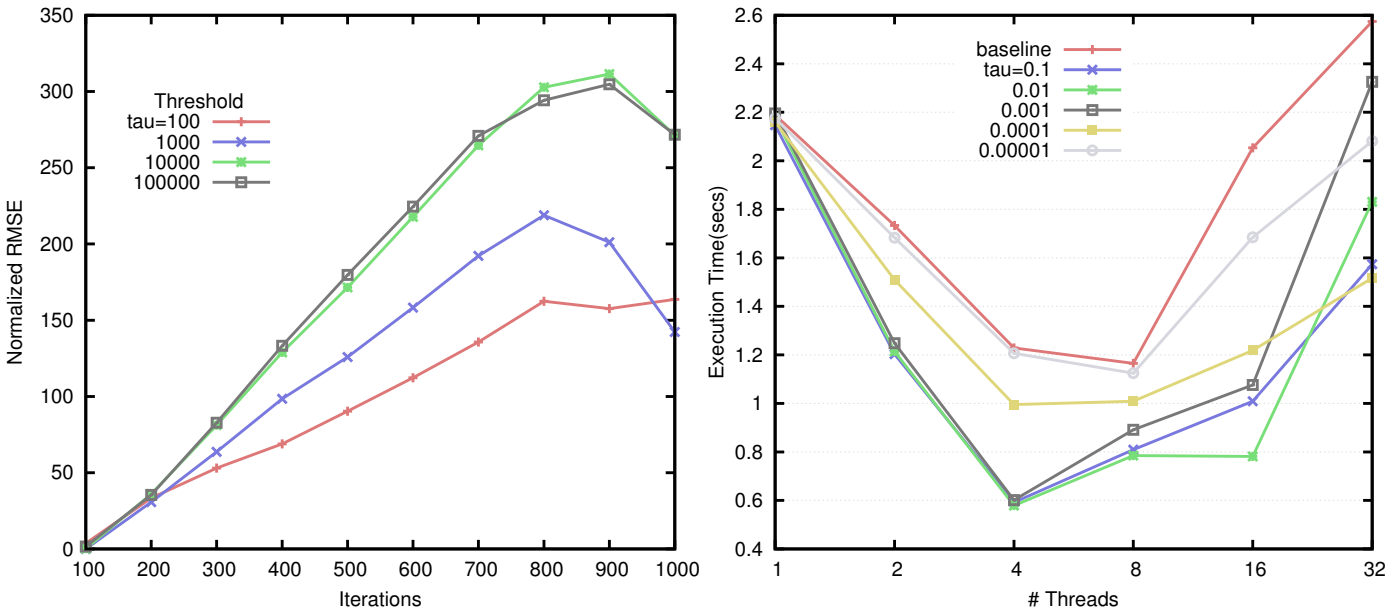
where f_p represents either the velocity or position of the particle p. The velocity $\overrightarrow{V_p}$ calculated for particle p in time-step $t+dt$ depends on the force $\overrightarrow{F_p}$ acting on the particle in time-step $t+dt$, which in turn depends on the distance vectors from particle p to all other particles within a *cutoff-distance* in time-step t. Additionally, the position $\overrightarrow{P_p}$ of a particle at time $t+dt$ depends on $\overrightarrow{V_p}$ at time t. This sharing of particle positions between threads is the main source of contention for this program. During each time step, new position and velocity vectors are computed for each particle. Depending on the granularity of the time step, the initial velocities and positions, the new vectors can differ from the old ones by very little. If this difference is so small that the old and new position vectors are *approximately-local*, then a consumer thread that consumes the old position vector for a particle need not abort if a new position vector is produced. Therefore a locality threshold can be applied on the shared position vectors to reduce contention among threads by avoiding weak false-conflicts. The performance impact of avoiding strong and weak false conflicts is shown in Figure 6b. The plot shows execution time using relative thresholds and the `REL` operation with τ ranging from 1E-1 to 1E-5. In most cases there is a substantial speedup with a maximum of 2.62x over the baseline.

Several previous works have identified key metrics in measuring the *fidelity* of a particle simulation. These include magnitude of error in linear and angular velocities, error in positions, error in energies etc. For our system the metric most relevant is particle position. In order to calculate error we first compute the Root Mean Square Error (RMSE) in particle positions relative to the outputs produced by the baseline. We then normalize this RMSE with the maximum size of the minimal box that contains all the particles. This is shown by the Normalized RMSE at the end of iteration 1000 in Figure 6a. This figure also shows the rate of growth of error during program execution. We see that this rate of growth is initially roughly linear and starts to reduce towards the end of the program. We also measured the distortion in the outputs produced by distinct baseline runs and we found that as in `kmeans` the RMSE was comparable to this distortion (roughly 40% more). This means that in a relative sense the mean error was comparable to what could be expected out of executions of the baseline program itself.

VIII. Related Work

A. Transaction Nesting

The topic of open nesting in software transactional memory systems has been studied extensively [20], [21]. The main



(a) Growth of error with MV threshold

(b) Execution time with REL threshold

Fig. 6: particle

Algorithm 4 `particle`

```
{/* Vector4D: pos, vel, mass, f */}
for time=0; time < NUM_STEPS; time += dt do
  for all {particle "i" ∈ ThisPartition } do
    atomic {
    for all {particle "j" ∈ NeighborWindow} do
      F[i] = computeForces(i, j)
      pos[i] = computePosition(pos[i], dt, vel[i])
    end for
    } endatomic
    vel[i] = computeVelocity(F[i], vel[i], mass[i])
  end for
end for
```

purpose of using open nesting is to separate physical conflicts from semantic conflicts since the programmer usually only cares about the latter. Therefore strict physical serializability is traded for *abstract serializability*. Abstract Nested Transactions [27] allow a programmer to specify operations that are likely to be involved in benign conflicts and which can be executed.

B. Silent Stores, Value Locality and Reuse

The phenomenon of silent stores has been extensively studied in the computer architecture community [19] and there have been numerous architectural optimizations suggested to exploit the same. Similarly, the phenomenon of *load value locality* has also been studied extensively [5]. Both these concepts basically establish that in many programs, values accessed by loads and stores tend to have a repetitive nature to them. In addition, techniques based on *value prediction* exploit the locality of values loaded in a program to apply optimizations such as cache prefetching. In [18] the authors explore the phenomenon of *frequent values* - values which collectively form the majority of values in memory at an instant during program execution. In [9], the STM system uses a form of value based conflict detection for improving performance. To our knowledge, this is the only STM system that is explicitly program value-aware. In [12], [7] the authors investigate the detection and bypassing of trivial instructions for improving performance and reducing energy consumption. Frameworks such as memoization [16], function caching [3] and value reuse [10] have been proposed to allow programs to reuse intermediate results by storing results of previously executed FP instructions and matching an instruction to check if it can be bypassed by reusing a previous result.

C. Relaxed Synchronization and Imprecise Computation

The idea of relaxed consistency systems has been studied in a few contexts. Zucker studied relaxed consistency and synchronization [25] from a memory model and architectural standpoint. In [1], the authors propose a weakly consistent memory ordering model to improve performance. In [2], the authors redefine and extend isolation levels in the ANSI-SQL definitions to permit a range of concurrency control implementations. In [4] the authors propose techniques to provide improved concurrency in database transactions by sacrificing guarantees of full serializability - weak isolation was achieved by reducing the duration for which transactions held read/write locks. A more recent work [8] work proposes Transaction Collection Classes that use multi-level transactions and open nesting, through which concurrency can be improved by relaxing isolation when full serializability is not required. In [22], the authors propose new programming constructs to improve parallelism by exploiting the semantic commutativity of certain methods invocations.

IX. Conclusions

A significant body of work exists on characterizing parallel applications in terms of design patterns, memory and cache behaviors, loop-level and task level parallelism and so on. However a set of significant questions remain largely unexplored: how do shared values in parallel soft-computing applications evolve, is it possible or desirable to synchronize

these values imprecisely, what are the accuracy-performance tradeoffs involved? With the rising ubiquity of soft-computing applications these and related questions merit exploration. In this paper we present the results of our investigation of these questions in the context of three representative workloads.

Conventional optimistic synchronization systems are designed to reason about meta-data of shared data in order to arbitrate conflicts. They consider a store operation as a production of a new value irrespective of the actual value being written. Consequently even if the written value is similar to the original value and the consumer of this value is tolerant of this approximation, it will be found to be in conflict. Hence existing techniques are severely limiting to the parallel performance that these applications can achieve. In this paper we presented the idea of Approximate Shared Value Locality and a technique to detect its occurrence. We also showed how this technique can be combined with a value based conflict arbitration mechanism to reduce the number of conflicts caused on *approximately local* values. We applied these techniques on a variety of workloads and found that a substantial reduction in abort rate and running time is possible while keeping the error introduced in the results small. In addition the rate of growth of error during execution was small in most cases. In future work we plan to investigate profiling and program analysis techniques that can help the programmer in estimating properties such as rate of growth of the error and the right threshold to use for a particular acceptable level error. It seems likely that these properties cannot be established in a domain-agnostic way or without some involvement from the programmer. Additionally we plan to extend these techniques to be able to reason about more complex program entities like pointers, compound data structures and arrays.

Although this paper discusses imprecise synchronization in the context of a software transactional memory system, the broad principles apply to other optimistic synchronization systems like speculative lock elison. Hence another interesting avenue for future work will be to explore and formulate a general framework that is independent of the specific underlying synchronization mechanism.

X. Acknowledgements

This paper is based upon work supported by the National Science Foundation under grant numbers CCF-0702286 and CCF-0916962

References

[1] Adve, S. V. and Hill, M. D. 1990. Weak orderinga new definition. In Proceedings of the 17th Annual international Symposium on Computer Architecture (Seattle, Washington, United States, May 28 - 31, 1990)

[2] Adya, A. 1999 Weak Consistency: a Generalized Theory and Optimistic Implementations for Distributed Transactions. Technical Report. UMI Order Number: TR-786., Massachusetts Institute of Technology.

[3] Heydon, A., Levin, R., and Yu, Y. 2000. Caching function calls using precise dependencies. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (Vancouver, British Columbia, Canada, June 18 - 21, 2000). PLDI '00

[4] Gray, J. N., Lorie, R. A., Putzolu, G. R., and Traiger, I. L. 1988. Granularity of locks and degrees of consistency in a shared data base. In Readings in Database Systems Morgan Kaufmann Publishers, San Francisco, CA, 94-121.

[5] Lipasti, M. H., Wilkerson, C. B., Shen, J. P. Value locality and load value prediction. SIGOPS Oper. Syst. Rev. 30, 5 (Dec. 1996), 138-147

[6] Minh C. C., Chung J., Kozyrakis C., Olukotun K., STAMP: Stanford Transactional Applications for Multi-Processing. IISWC 2008: 35-46

[7] Richardson S.E., Exploiting Trivial and Redundant Computation, Sun Microsystems Technical Report 1993.

[8] Ni, Y., Menon, V. S., Adl-Tabatabai, A., Hosking, A. L., Hudson, R. L., Moss, J. B., Saha, B., and Shpeisman, T. 2007. Open nesting in software transactional memory. In Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (San Jose, California, USA, March 14 - 17, 2007)

[9] Olszewski, M., Cutler, J., and Steffan, J. G. 2007. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In Proceedings of the 16th international Conference on Parallel Architecture and Compilation Techniques (September 15 - 19, 2007)

[10] Citron D. and Feitelson D. G.,"Look It Up" or "Do the Math": An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization, in Workshop on Power Aware Computing Systems 2004.

[11] Dice D., Shalev O., Shavit N., Transactional Locking II. Proceedings of the 20th International Symposium on Distributed Computing (DISC), 2006.

[12] Richardson S. E., Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation Sun Microsystems Technical Report TR-92-1 1992.

[13] Zadeh L.A., Fuzzy logic, neural networks, and soft computing. Communications of the ACM, 37(3):7784, 1994.

[14] Shavit, N., Touitou, D. Software transactional memory. PODC 95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (New York, NY, USA, 1995), ACM Press, pp. 204213.

[15] De Gelas, J. The quest for more processing power: Multi-core and multi-threaded gaming. http://www.anandtech.com/cpuchipsets/showdoc.aspx, March 2005.

[16] Acar U. A., Blelloch G. E., and Harper R. 2003. Selective memoization. SIGPLAN Not. 38, 1 (Jan. 2003)

[17] Baek W., Chung J., Minh C. C., Kozyrakis C., and Olukotun K., Towards soft optimization techniques for parallel cognitive applications. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (San Diego, California, USA, June 09 - 11, 2007). SPAA '07. ACM, New York, NY, 59-60.

[18] Yang, J. and Gupta, R. 2002. Frequent value locality and its applications. Trans. on Embedded Computing Sys. 1, 1 (Nov. 2002)

[19] Lepak, K. M., Exploring, Defining, and Exploiting Recent Store Value Locality. Ph.D. thesis. The University of Wisconsin-Madison, Department of Electrical and Computer Engineering. Dec. 2003.

[20] Ni Y., Menon V., Adl-Tabatabai A. R, Hosking A.L, Hudson R.L., Moss J.E.B., Saha B., Shpeisman T., Open nesting in software transactional memory. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP):68-78 March 2007.

[21] Eliot, J., Moss, B., Open nested transactions: Semantics and support. In Workshop on Memory Performance Issues 2005.

[22] Kulkarni M., Pingali K., Walter B., Ramanarayanan G., Bala K. and Chew L. P. 2007. Optimistic parallelism requires abstractions. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (San Diego, California, USA, June 10 - 13, 2007)

[23] Sims K., Particle Animation and Rendering Using Data Parallel Computation. In Proceedings of SIGGRAPH 1990

[24] Yeh T. Y., Reinman G., Patel S. J., and Faloutsos P. 2009. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. ACM Trans. Graph. 29, Dec. 2009

[25] Zucker, R. N. Relaxed consistency and synchronization in parallel processors. Tech. Rep. TR-92-12-05, University of Washington, 1992.

[26] Yeh T., Faloutsos P., Ercegovac M., Patel S. and Reinman G. 2007. The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration. In Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture (December 01 - 05, 2007).

[27] Harris T. and Stipic S., "Abstract Nested Transactions", in 2nd Workshop on Transactional Computing (TRANSACT 07)

Improving Virtualization Performance and Scalability with Advanced Hardware Accelerations

Yaozu Dong*, Xudong Zheng*, Xiantao Zhang*, Jinquan Dai*, Jianhui Li*, Xin Li*, Gang Zhai*， Haibing Guan#

*Intel China Software Center, Shanghai, China
{eddie.dong, xudong.zheng, xiantao.zhang, jason.dai, jian.hui.li, xin.li, edwin.zhai}@intel.com

#Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China
hbguan@sjtu.edu.cn

Abstract — Many advanced hardware accelerations for virtualization, such as Pause Loop Exit (PLE), Extended Page Table (EPT), and Single Root I/O Virtualization (SR-IOV), have been introduced recently to improve the virtualization performance and scalability. In this paper, we share our experience with the performance and scalability issues of virtualization, especially those brought by the modern, multi-core and/or overcommitted systems. We then describe our work on the implementation and optimizations of the advanced hardware acceleration support in the latest version of Xen. Finally, we present performance evaluations and characterizations of these hardware accelerations, using both micro-benchmarks and a server consolidation benchmark (vConsolidate). The experimental results demonstrate an up to 77% improvement with these hardware accelerations, 49% of which is due to EPT and another 28% due to SR-IOV.

Keywords: Virtualization, Virtual Machine; PLE, EPT, SR-IOV

I. INTRODUCTION

Virtualization is the key enabling technology for server consolidation and elastic resource management in the cloud environment [1]. However, virtualization not only brings additional performance overheads (in CPU and memory, as well as I/O), but also introduces additional scalability issues for multi-core and/or overcommitted systems.

Full virtualization (with binary translation) [25] and paravirtualization [6][9][28] were proposed when there were not sufficient hardware supports. Full virtualization supports unmodified guest OSes, but requires the OS binary to be translated at run time. Paravirtualization improves virtualization performance with moderate modifications to OS codes, which, however, is difficult to apply to proprietary OSes. Basic hardware virtualization technologies, such as Intel Virtualization technology [23], are introduced to eliminate the need for binary translation in full virtualizations and to improve the virtualization performance.

Unfortunately, these traditional approaches are inefficient in or incapable of addressing the virtualization performance and scalability issues, especially those brought by the modern, multi-core and/or overcommitted systems. To further improve the performance and scalability of virtualization, a lot of advanced hardware accelerations are introduced, such as Pause Loop Exit (PLE) [12], Extended Page Table (EPT) [12], and Single Root I/O Virtualization (SR-IOV) [18].

In this paper, we present our work of supporting these advanced hardware accelerations in the latest version of Xen [6], sharing our experience on how these accelerations are used and what additional software optimizations are required to improve the virtualization efficiency. The major contributions of our work are as follows.

- Summary of our experience on the performance and scalability issues of virtualization, especially those brought by the modern, multi-core and/or overcommitted systems.
- Presentation of our work on the implementation and optimizations of advanced hardware acceleration support in the latest version of Xen, including hybrid interrupt vectoring, improving page table locality with large memory pages and confined Guest Page Table, and empirical PLE configuration.
- Performance evaluations and characterizations using not only micro-benchmarks, but also a server consolidation benchmark (vConsolidate) [3].

The results show an up to 77% performance improvement (49% of which due to EPT and another 28% due to SR-IOV) in the server consolidation benchmark, and an up to 14% improvement, due to PLE in the micro-benchmarks.

The rest of the paper is organized as follows. Section II provides an overview of the virtualization challenges (brought by the modern, multi-core and/or overcommitted systems). Section III describes the advanced hardware accelerations for virtualization and how they are supported in Xen. Section IV describes the required software optimizations when using the hardware accelerations. Section V presents the characterizations and improvements with the advanced hardware accelerations for vitalization, using both micro-benchmarks and a server consolidation benchmark. Section VI discusses the related work and finally section VII concludes the paper.

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

II. Challenges of Virtualization Performance and Scalability

In this section, we first provide an overview of the virtualization architecture of Xen, and then describe the challenges of virtualization performance and scalability (especially those brought by the modern, multi-core and/or overcommitted systems).

A. Xen Architecture

Figure 1 shows the virtualization architecture implemented in Xen, based on Xen's Hardware Virtual Machine (HVM) implementation with basic hardware supports such as Intel Virtualization Technology [23].

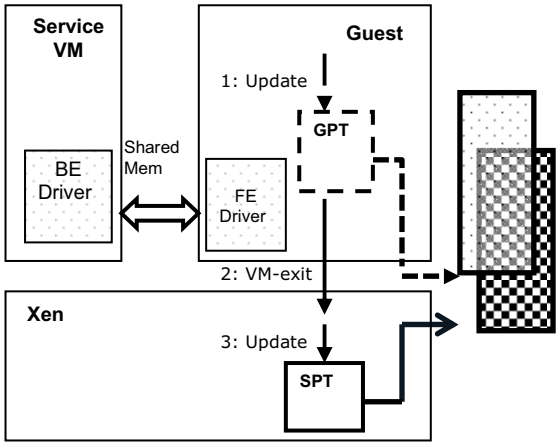


Figure 1. Xen HVM architecture

Shadow Page Table (SPT) is adopted in Xen HVM to virtualize memory. SPT is fully controlled by Xen hypervisor to manage the translation from guest linear memory address to host physical memory address. The guest VM only updates the Guest Page Table (GPT), which is write-protected so that the update of the GPT entry by the guest can be trapped-and-emulated (through the VM-exit [23] event handler in the hypervisor). Consequently, it can be guaranteed that the SPT is always in synchronization with the GPT.

Since full I/O virtualization incurs extremely high overheads due to excessive hypervisor interventions, I/O paravirtualization (or PV I/O) [9] is adopted in Xen HVM. However, in PV I/O the hypervisor still needs to intervene in an I/O request from the frontend in the guest VM and activate the backend driver in the service VM (that is domain 0 in Xen), which then responds to the request (such as receiving a packet from device and copying the packet to the guest buffer).

B. Virtualization challenges

Unfortunately, both SPT and PV I/O bring additional costs for virtualization. In particular, SPT suffers from excessive VM-exit (caused by the write-protection on GPT). The performance of PV I/O is limited by the interaction between the frontend and backend drivers and the packet copy in the service VM.

In addition, multi-processor (MP) guest OS and overcommitted systems bring new challenges to the virtualization performance and scalability. For instance, MP kernel usually has very heavy usage of locks to ensure atomic accesses to shared data structure. In a system where virtualized CPUs are overcommitted, the impacts of lock contentions are even worse because the VCPU that currently holds a lock may happen to be preempted by the hypervisor [24][29].

MP guest also brings additional synchronization overheads due to lock contentions on Shadow Page Table accesses. The hypervisor needs to track the page table for each VCPU. Because MP guest shares the page table entries for the kernel space among VCPUs, the corresponding SPT entries are also shared among these VCPUs to avoid the synchronization effort, and consequently the hypervisor needs to first acquire a lock before it updates the SPT.

Lastly, PV I/O has serious scalability issue in an overcommitted system [14][16][20][22]. As the guest VM count increases, multiple frontend drivers will request services from a single backend driver concurrently, which becomes the scalability bottleneck (such as saturating the VCPU of the service VM due to the packet copy).

III. Advanced Hardware Accelerations

In this section, we describe the advanced hardware accelerations and how we extended the Xen architecture to support them, addressing the performance and scalability challenges of virtualization. Figure 2 shows the architecture of the latest version of Xen that supports these hardware accelerations.

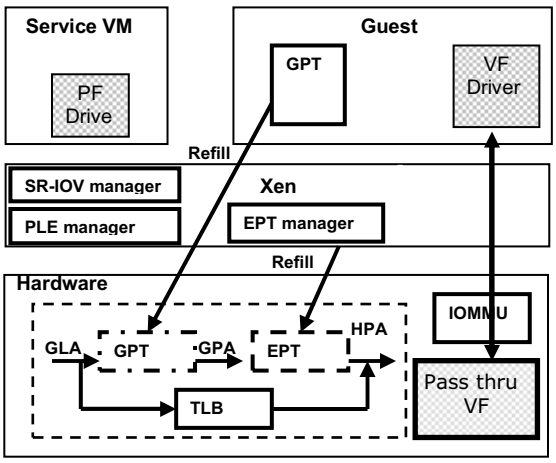


Figure 2. Extended Xen architecture using advanced hardware accelerations

A. Pause Loop Exit

Pause Loop Exit (PLE) technology provides hardware assistance to detect guest spin-lock. Modern CPU employs PAUSE instruction as spin loop hint to the processor. However, the PAUSE instruction is also widely used in the OS kernel for other purposes (such as in a delay loop). PLE

technology enables VMM to detect spin-lock by monitoring the execution of PAUSE instructions through the PLE_Gap and PLE_Window values. PLE_Gap is an upper bound on the cycle count between two successive executions of PAUSE in a loop. The CPU detects a spin-lock if it identifies a series of PAUSE instruction executions and the latency between two successive executions is within the PLE_Gap value. PLE_Window is an upper bound on the amount of time a VCPU is allowed to execute in a spin-loop. If a spin loop execution time surpasses the PLE_Window value, the CPU will invoke a VM-exit to notify the hypervisor.

The PLE manager in Xen yields its CPU quantum when receiving a VM-exit indicating a long-waiting spin loop. In addition, the PLE manager is responsible for setting the PLE_Window and PLE_Gap values.

B. *Extended Page Table*

The Extended Page Table (EPT), which implements a two-dimensional page table [4], is proposed to eliminate the excessive VM-exit and lock contentions in Shadow Page Table. With the EPT support, the Guest Page Table (GPT) contains the mapping from the guest linear memory address (GLA) to guest physical address (GPA), while the EPT contains the mapping from guest physical address to host physical address (HPA), as illustrated in Figure 3.

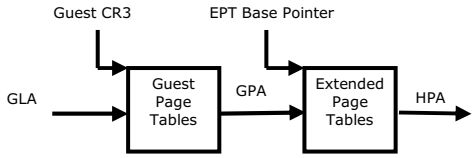


Figure 3. Extended Page Table (EPT)

The EPT manager in XEN maintains an EPT table per guest, which works with the EPT hardware to manage the translation from GPA to HPA, as shown in Figure 2. The guest in EPT gets back the ownership of its page tables, and consequently EPT can completely eliminate the need for the write-protection on Guest Page Table accesses and avoids altogether the excessive VM-exit and lock contentions.

Address translation in the EPT is accelerated by paging-structure caches and Translation Look-aside Buffer (TLB). With the EPT, hardware walks both the guest page table and the EPT, fills the paging-structure caches and forms a merged TLB, translating from the GLA to the HPA directly, as shown in Figure 2.

C. *SR-IOV*

SR-IOV proposes a set of hardware enhancements to the PCIe device [18]. An SR-IOV-capable device has single or multiple Physical Functions (PFs). Each PF can create multiple "light-weight" instances of PCI function entities, known as Virtual Functions (VFs), which are configured and managed by the PF. Each VF can be assigned to a guest for direct access, but still shares major device resources, to achieve both resource sharing and high performance (offloading memory protection and address translation to IOMMU).

We implemented a generic virtualization framework for SR-IOV devices, as shown in Figure 2. The PF driver running in the service VM is responsible for configuring and managing the VFs. On the other hand, the VF driver running in the guest has direct access to the VF runtime resource. The SR-IOV manager receives an interruption from the I/O device, and redirects it to the VF driver in the guest, which can then access the associated data directly from the VF resources. In particular, our implementation with the SR-IOV-capable NIC eliminates the well-known packet classification, address translation and copy overhead, which has demonstrated a 2.63X (from 3.6Gbps to 9.48Gbps) performance improvement over PV NIC [8].

IV. SOFTWARE OPTIMIZATIONS

In this section, we share our experience with these advanced hardware accelerations, and describe what additional software optimizations are required to improve the virtualization efficiency when using the hardware accelerations.

A. *Hybrid Interrupt Vectoring*

The scalability of SR-IOV suffers from the shortage of interrupt vectors. Each VF interrupt source in Xen SR-IOV needs to own a dedicated vector to avoid interrupt sharing. However, the number of interrupt vectors in the x86 architecture is limited to 256, and some of them are reserved or pre-allocated by system. Consequently, the number of interrupt vectors available for VFs is only around 200. However, each VF in an SR-IOV-capable device (such as Intel 82599 NIC [13]) may consume 3 interrupt sources. Conseqeuntly, Xen can only accommodate about 66 VFs before the vectors are exhausted.

To address this issue, we have implemented per-PCPU vectoring in the latest version of Xen. In per-PCPU vectoring, an interrupt is represented using both a PCPU ID and a vector number. When the interrupt is received, hypervisor remaps the physical interrupt to guest interrupt with a per-PCPU remap table before it fires the guest interrupt to the mapped VCPU. Consequently, total number of interrupt vectors that can be accommodated in the system can be increased by multiple-fold, which greatly improves the SR-IOV scalability.

However, in per-PCPU vectoring, I/O APIC (Advanced Programmable Interrupt Controller) may generate redundant messages for level-triggered interrupts. This is because after delivering a level-triggered interrupt message, the I/O APIC will suppress further delivering of the same interrupt until an End of Interrupt (EOI) message is issued by the CPU. However, the EOI message is not tagged with the PCPU ID (due to the current hardware implementations). Consequently, when an EOI message is received, the I/O APIC will stop the suppression of all the interrupts with the

matching vector number, even though the EOI message may be issued by a completely different PCPU. So, the I/O APIC may generate redundant interrupt messages for level-triggered interrupts.

Our hybrid vectoring mechanism resolves this issue by using global vectoring for level-triggered interrupts, which are the minority of interrupt sources due to the limitation of I/O APIC pin #, and per-CPU vectoring for edge triggered interrupts, which are the majority of interrupt sources in modern platform (Message Signaled Interrupts fall into this category).

B. *Improving Page Table Locality*

With EPT, hardware needs to walk both the Guest Page Table and EPT to fetch both table entries, to form a merged TLB, translating from the guest linear memory address to the host physical address directly. Unfortunately, this introduces additional cycles to walk the EPT table to refill a TLB entry. In particular, a single 4-level Guest Page Table walk invokes 5 rounds of the 4-level EPT walk, one for each guest page entry and an additional one for the final translation of the guest physical address of the datum itself [4].

The additional EPT walk may cost non-trivial CPU cycles if cache miss happens frequently during the page table walk, and therefore, it is critical to improve the locality of the TLB entries, EPT paging-structure caches, and memory caches. We have implemented large memory pages in Xen by allocating the physical memory for each guest in 2MB chunks. Consequently, the last level EPT page table entry (EPTE) can be completely by-passed in EPT walk. In addition, large memory pages help reduce TLB pressures because each TLB entry can now cover more address space.

Furthermore, guest OSes could be paravirtualized to allocate the memory pages for the Guest Page Table from one or more contiguous 2MB physical memory spaces (that is, confining Guest Page Table in one or more 2MB working sets for each guest process), which can further improve the cache locality of the EPT walk (likely hitting the same EPT entries for all Guest Page Table walk).

C. *Empirical PLE Configurations*

The configuration of PLE_Gap and PLE_Window in PLE_manager is critical for identifying guest spin-lock and generating VM-exit events. The VM-exit event helps the hypervisor to perform better scheduling. However, overly frequent VM-exit can also introduce additional virtualization overheads.

We have used an empirical approach to determine these two values. Figure 4 shows the lock acquisition time in Kernel Build with a 16 VCPUs HVM guest running on top of 16 PCPUs. There are only 0.5% of all spin-locks that may wait more than 2048 cycles. Therefore, we choose to set PLE_Window (the maximum cycles a VCPU is allowed to execute a spin-lock loop continuously) to 2048 cycles. PLE_Gap is not sensitive to performance as if it is bigger than a certain threshold as shown in Figure 5. We set PLE_Gap to a moderate value (128 cycles), that is, if two successive executions of the PAUSE instruction occur within 128 cycles, they are considered to be from the same spin-lock loop.

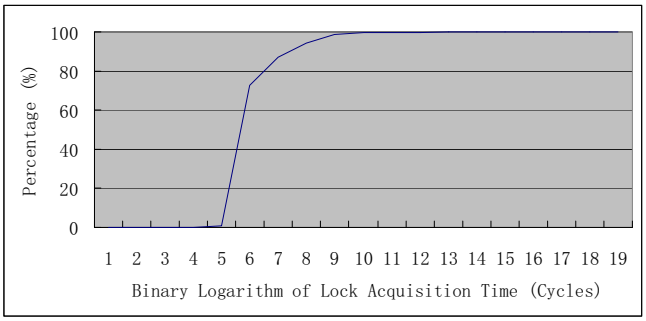


Figure 4. Lock acquisition time in Kernel Build

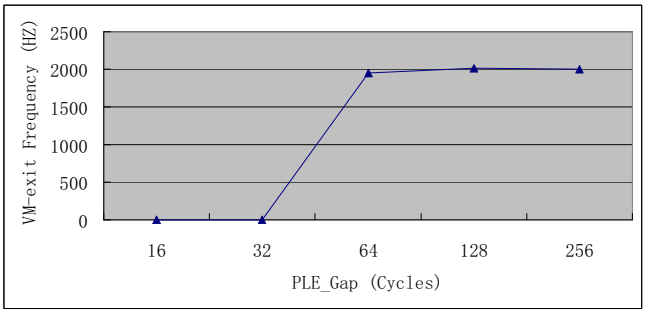


Figure 5. VM-exit frequency of PAUSE instruction in Kernel Build

V. EVALUATIONS

In this section, we present the characterizations and improvements of virtualization and scalability, for the advanced hardware accelerations and corresponding software optimizations. Our experiment runs a server consolidation benchmark (vConsolidate), as well as micro-benchmarks on an Intel Xeon server system. The detailed configurations of the system used in our experiment are shown in Table I.

TABLE I. SYSTEM CONFIGURATIONS

Host Processor	Dual-socket quad-core Intel Xeon processor (SMT enabled)
Host Memory	48GB ECC DRAM
Host Storage	Storage Area Network (SAN)
Host Network	Intel 82599 10GbE NIC (with SR-IOV support)
Hypervisor	Xen Upstream (64bit, change set 19590)
HVM Guest	RHEL 5U1 (kernel updated to 2.6.30)

The micro-benchmark results show that PLE brings up to 14% improvements. EPT brings about 24% performance improvements and achieves 3.9X speedup over Shadow Page Table in the UP guest and 16-VCPU MP guest respectively. Large memory pages and confined Guest Page Table bring ~2% and ~3% additional performance improvements, respectively. In the server consolidation benchmark, EPT achieves an up to 49% performance advantage over Shadow Page Table. SR-IOV brings an additional 28% improvement.

In total, EPT+SR-IOV can reach about 1.77x the performance of that in the baseline Shadow Page Table test.

A. Micro-Benchmarks

We used several micro-benchmarks (including Kernel Build, SPECjbb, WebBench, and Netperf) in our experiment. When running WebBench and Netperf, the testing system runs as the "server" and is directly connected to the "client" that runs in native. The service VM (that is domain 0 in Xen) employs 8 VCPUs. The other detailed configurations of the VM guests for these micro-benchmarks are shown in Table II.

TABLE II. MICRO-BENCHMARK GUEST CONFIGURATIONS

	Guest OS	VCPUs	Guest Memory
Kernel Build	32bit Linux	1-64	512MB
SPECjbb	64bit Linux	2	4GB
WebBench	32bit Linux	2	1.5GB
Netperf	32bit Linux	1	512MB

- **Kernel Build**

Figure 6 shows the overheads of Shadow Page Table (SPT) in Kernel Build. In the uniprocessor (UP) guest, about half of the VM-exit events are due to the write-protection on Guest Page Table (GPT), and about 12.5% of the total CPU cycles are spent on processing these VM-exit. On the other hand, in the multi-processor guest (4 VCPUs in our experiment), though the VM-exit events caused by the write-protection on GPT are fewer than those in the UP guest, about 18.7% of the total CPU cycles are spent on processing these VM-exit. This is because the lock contentions on SPT accesses in the hypervisor brings extra synchronization overheads.

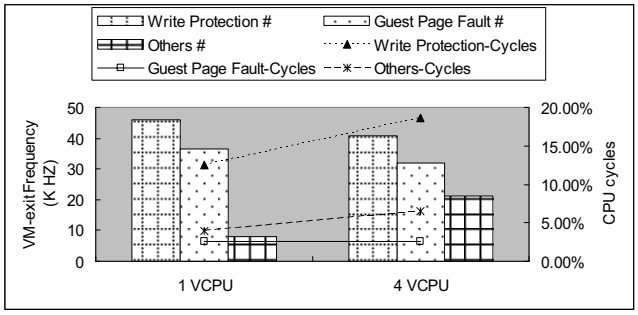


Figure 6: Overheads of Shadow Page Table in Kernel Build

Figures 6 and 7 show the performance and scalability improvements brought by the Extended Page Table (EPT) and large memory pages. In the UP guest (as shown in Figure 7), we run Kernel Build in each guest and measure the average time spent per build. EPT brings about 24% performance improvements over Shadow Page Table, and large memory pages bring about 2% additional performance improvements. In the MP guest (as shown in Figure 8), the performance of Shadow Page Table does not scale beyond 4 VCPUs (due to the lock contentions in the hypervisor), while EPT scales well, even with 16 VCPUs (about 8.17X speedup compared to the 1-VCPU case). EPT achieves 3.9X speedup compared to the SPT in 16-VCPU case.

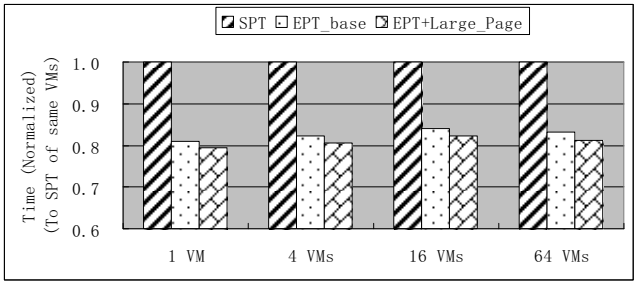


Figure 7. Performance and scalability of Kernel Build in multiple UP guests

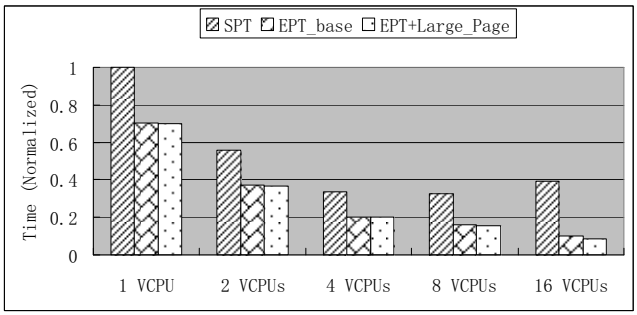


Figure 8. Performance and scalability of Kernel Build in an MP guest

Figure 9 shows the performance and scalability improvements brought by Pause Loop Exit (PLE) on top of EPT with large memory pages when running Kernel Build in the MP guest with 4 VCPUs. PLE brings about 14% improvements when the virtualized CPUs are overcommitted (that is 16-VM case with a total of 64 VCPUs), but it doesn't help if the CPU is not overcommitted (that is 1-VM case with a total of 4 VCPUs).

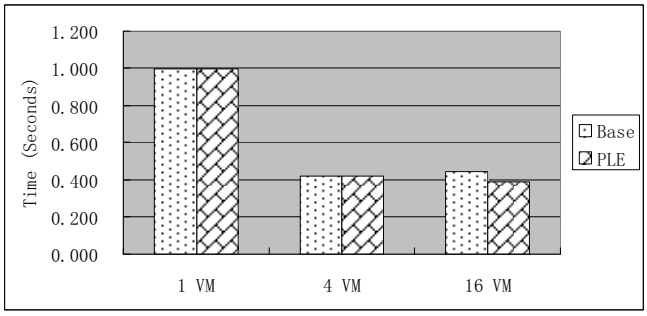


Figure 9. Performance and scalability of Kernel Build in a 4-VCPU guest

- **SPECjbb/WebBench**

EPT with large memory pages achieves 8% and 14% performance advantage over base line EPT in SPECjbb and WebBench, respectively. WebBench is more sensitive to large memory pages than SPECjbb, because WebBench frequently switches its processes (and hence page tables) per client requests, stressing the TLB entries, EPT paging-structure caches, and memory caches. The results are shown in Figure 10.

The benefits of confining Guest Page Table to one or more contiguous 2MB physical memory space depend on the

memory allocation patterns of the applications. If the application allocates memory pages in a relatively static fashion, the memory pages of the Guest Page Table typically come from a contiguous memory block (such as in Linux). However, when the application frequently allocates and frees memory pages, the memory pages of the Guest Page Table may scatter in different locations across the entire memory space. Consequently, Guest Page Table confinement will bring bigger performance gains.

Figure 10 also shows that confining Guest Page Table to a 2MB contiguous physical memory space achieves ~3% additional performance advantage on top of EPT with large memory pages in WebBench, which frequently create and destroy pages per client requests. However, there are almost no performance gains for SPECjbb, because it allocates memory in a relatively static fashion.

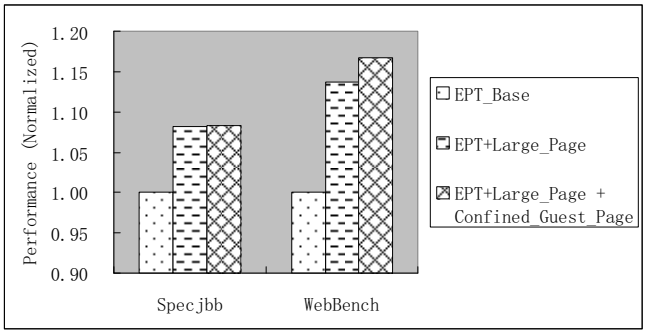


Figure 10. Performance of SPECjbb/WebBench with Confined Guest Page Table

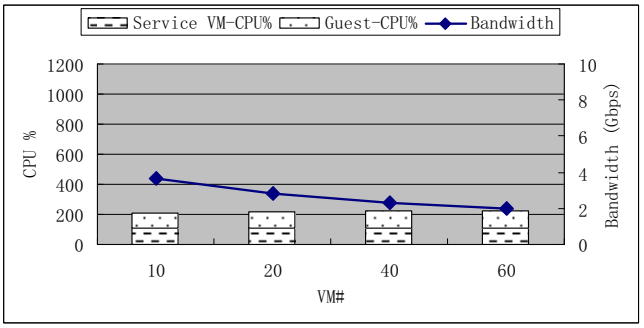


Figure 11: Performance of single-threaded PV NIC (100% CPU utilizations means a full thread cycles)

- **Netperf**

The scalability of PV NIC suffers from excessive CPU utilizations due to packets copy in service VM. The existing Xen PV NIC driver uses only a single thread in the backend for packet copy, which can easily saturate at 100% CPU utilization (although the tested system and the service VM has 16 threads in total, that is 1600% CPU resource). Consequently, it does not scale well with additional CPU cores and can achieve only about 3.6Gbps peak bandwidth in our experiment (using 10 VM guests), as shown in Figure 11.

We have enhanced the Xen PV NIC driver to run in multi-thread mode for backend service, to improve its scalability. Consequently, the enhanced PV NIC can achieve the line rate (that is 9.48 Gbps) with 10 VM guests at the cost of more than 700% CPU overheads, as shown in Figure 12.

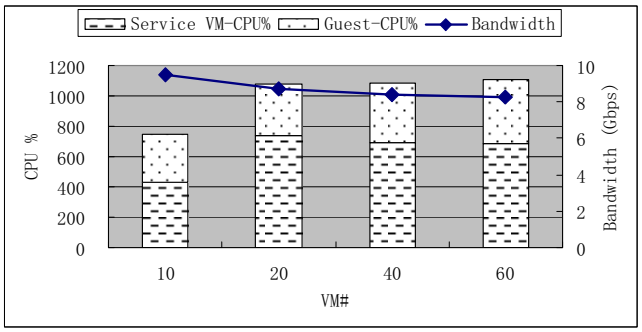


Figure 12: Scalability of multi-threaded PV NIC (100% CPU utilizations means a full thread cycles)

In SR-IOV test, the service VM runs the PF driver and a dedicated VF is assigned to each guest. As shown in Figure 13, SR-IOV demonstrates perfect scalability, achieving line rate (that is 9.48Gbps) with 60 VM guests at the cost of only 0.88X additional CPU thread overhead over 10 VM guests.

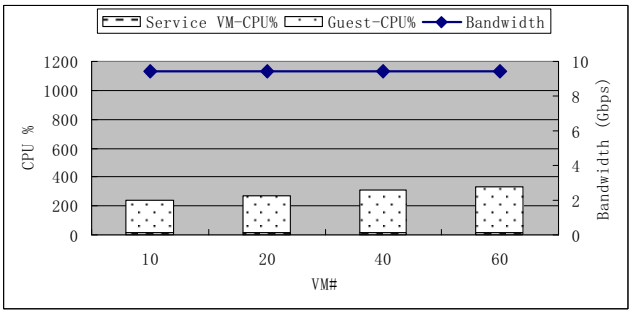


Figure 13. Scalability of SR-IOV NIC (100% CPU means a full thread cycles)

B. Server Consolidation Benchmark

vConsolidate [3] is a VMM agnostic benchmark for virtualization consolidation, which includes a compute-intensive workload, a Web server, an email server and a database server. Each workload runs in a separate VM. The compute-intensive workload runs SPECjbb, which is modified to consume roughly 50% of the CPU, by inserting random sleep sessions every few milliseconds to represent the real life scenario. The Web server runs WebBench using Apache Webserver. The email server runs LoadSim, representing a Microsoft Exchange workload that runs transactions on Outlook with 500 users logged-in simultaneously. The database server runs SysBench, representing an OLTP workload running transactions against an MySQL database. An idle VM is added to mimic the real life scenario because datacenters are not fully used at all times. Those 5 VMs running different workloads comprise a Consolidated Stack Unit (CSU).

The throughput of each CSU is measured as a geometric

mean of all 4 workloads (Web, Mail, DB, and Java) and the total system throughput is a sum of all the CSUs the system runs.

TABLE III. VCONSOLIDATE CONFIGURATIONS

	VCPUs	Guest Memory	Guest OS	Application
Web	2	1.5GB	32-bit Linux	Apache
Mail	1	1.5GB	32-bit Windows	Exchange
DB	2	1.5GB	64-bit Linux	MySQL
Java	2	2.0GB	64-bit Linux	BEA JVM
Idle	1	0.4GB	32-bit Windows	

In our experiment, we run vConsolidate using the configuration shown in Table III. The service VM employs 4 VCPUs with 512MB memory and all guests use PV disk and PV NIC drivers, except that in the SR-IOV test, each guest runs a VF driver with a dedicated VF. In the server, or system under test (SUT), each CSU uses 2 Intel X25E 64GB SSDs as its storage, and talks to a mail client, a WebBench client, and a Web controller (as shown in Figure 14). Multiple CSUs are used in our experiment, and the server and clients are connected through a Gigabit Ethernet switch.

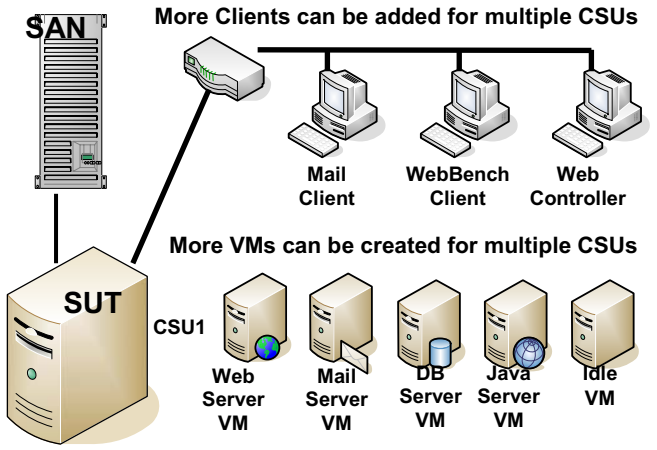


Figure14. The vConsolidate experiment environment

- **Performance**

Figure 15 shows the vConsolidate performance in the 1-CSU configuration. EPT+Large_Page, or EPT_LP configuration supports both hardware EPT support and large memory pages optimization, achieves about 29% performance improvement over Shadow Page Table (SPT) with lower CPU utilizations, and large memory pages in EPT bring 3% performance gain over EPT_base, which only uses hardware EPT support. Using SR-IOV further reduces the CPU utilizations from 28% to 24%. An interesting observation is that PV I/O has a 1.5% performance advantage over SR-IOV. This is because the CPU is not overcommitted in the 1-CSU configuration, and packets copied by the spare CPUs in service VM may improve cache locality.

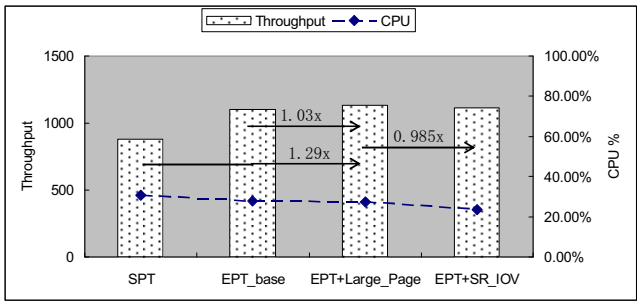


Figure15. vConsolidate performance in the 1-CSU configuration

Further breakdown of vConsolidate performance is shown in Figure 16. WebBench performance in EPT_LP is about 215% of that in SPT, which dominates the contribution to the overall vConsolidate performance. SPECjbb, SysBench, and LoadSim demonstrate moderate performance improvement (up to 14%).

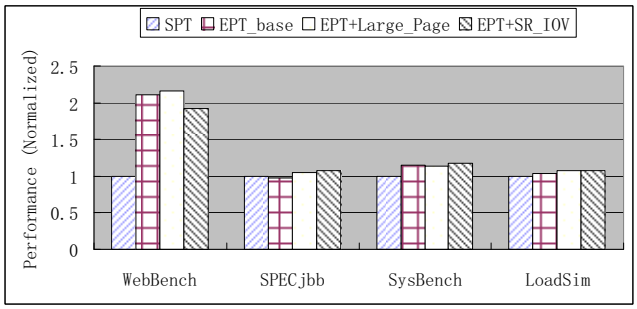


Figure 16. Breakdown of vConsolidate performance

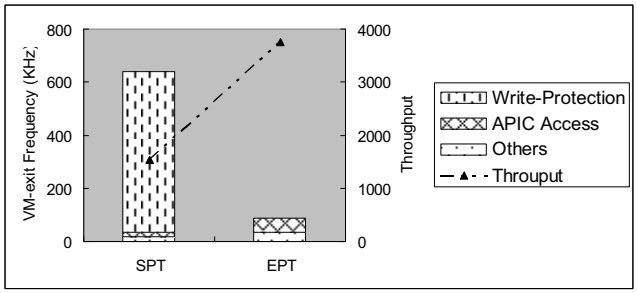


Figure17. VM-exit Frequency in WebBench

The high performance boost of WebBench in EPT_LP is due to the elimination of excessive VM-exit caused by the write-protection on Guest Page Table. As shown in Figure 17, these VM-exit events account for about 95% of the total VM-exit events in WebBench, and about 38% of the virtualization overhead of total CPU cycles. Most of these VM-exit events actually come from the execution of fork function, which reflects the fact that WebBench uses fork frequently to create a new process to handle each client request.

- **Scalability**

The performance of vConsolidate does not scale well in Shadow Page Table when more Consolidation Stack Units (CSUs) are deployed, as shown in Figure 18. Shadow Page

Table (SPT) achieves about 58% performance improvement from 1 to 2 CSUs, and tops its performance at the 3-CSUs configuration with an additional 23% improvement, using 95% CPU cycles. The SPT test fails to run vConsolidate with the 4-CSU configuration due to mail server failure.

The performance of EPT_LP scales much better than that of Shadow Page Table. The EPT_LP performance improves by 75% from 1 to 2 CSUs, and by 24% improvement from 2 to 3 CSUs (with 94% CPU utilizations). EPT_LP tops its performance in the 4-CSU configuration with 99% CPU utilizations, with an additional 3% performance gain over the 3-CSU configuration. The peak performance of the EPT_LP in the 4-CSU configuration is 2.25x of that in the 1-CSU configuration, and about 1.49x of the peak performance of Shadow Page Table (in the 3-CSU configuration).

Using SR-IOV together with EPT_LP achieves even better scalability, as shown in Figure 18. It brings about 3%, 8%, and 19% performance advantages over EPT_LP only, in the 2-, 3- and 4-CSU configurations, respectively. Its peak performance in the 4-CSU configuration is 2.73x of that in the 1-CSU configuration, and about 1.77x of the peak performance of SPT (an additional 28% improvement compared to EPT_LP only). This is due to the elimination of the service VM intervention in the guest network packet processing, because the intervention introduces additional network latency, service VM overhead, and cache pollution.

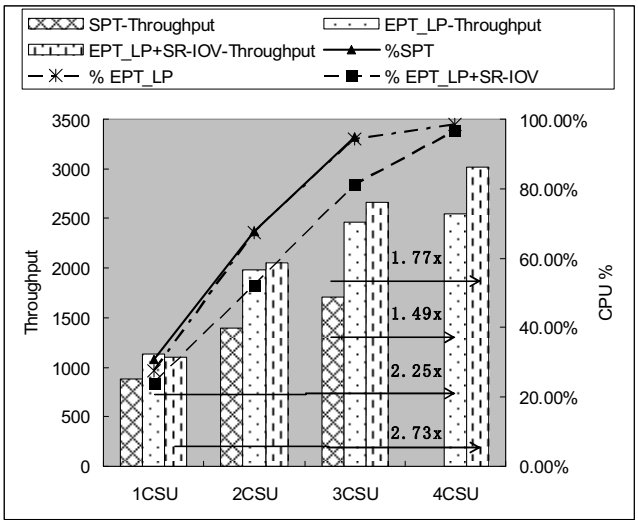


Figure 18. vConsolidate scalability

Further breakdown of scalability of each workload in SPT, EPT_LP and EPT_LP+SR-IOV are shown in Figures 19 – 21. With both EPT_LP and SR-IOV, all workloads except SysBench scale very well from 1 to 4 CSUs, before the CPUs are saturated. SysBench have very heavy disk I/O and reaches its peak performance in the 3-CSU configuration.

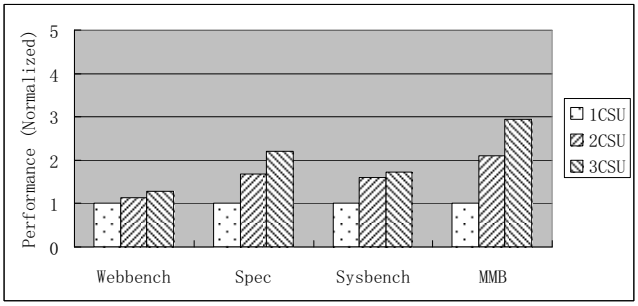


Figure 19. Breakdown of vConsolidate Scalability in Shadow Page Table

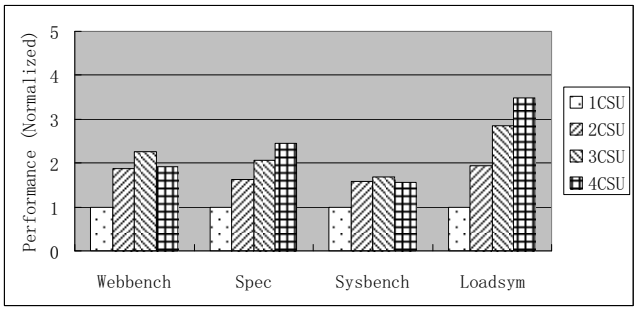


Figure 20. Breakdown of vConsolidate Scalability in EPT_LP

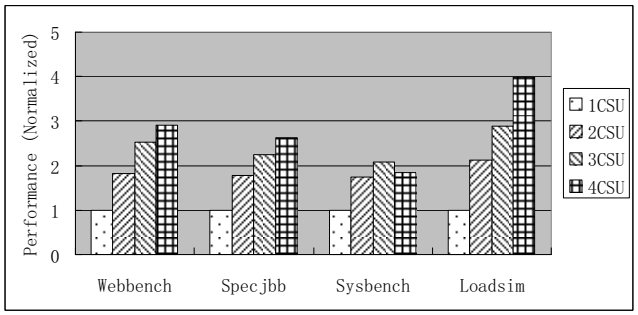


Figure 21. Breakdown of vConsolidate Scalability in EPT_LP+SR-IOV

The EPT_LP test has similar scalability, but it has higher CPU utilizations than when SR-IOV is also used. WebBench in EPT_LP configuration reaches its peak performance in the 3-CSU configuration, because it has very high network I/O and PV NIC brings very high overheads in the 4-CSU configuration. On the other hand, Shadow Page Table cannot scale beyond 3 CSUs because of the excessive VM-exit and lock contentions.

VI. Related Work

Full virtualization (with binary translation) [25], proposed when there were not sufficient hardware supports, supports unmodified guest OSes, but requires the OS code to be translated to binary when needed. Unfortunately, this incurs very high virtualization overheads. Basic hardware virtualization technologies, such as Intel Virtualization technology [23], are introduced to eliminate the need for binary translation and to improve the virtualization performance. In addition, paravirtualization [6][9] improves virtualization performance with moderate modifications to OS codes, which however is difficult to apply to proprietary

OSes.

Impacts of spin-lock to MP guest performance and scalability have been explored before. For instance, the approaches to donate the wasted spinning time to the lock holder or to avoid preempting lock-holders are studied in [24]. Heuristic based detection of lock-holder, when the number of unique stores executed within N committed instructions is less than some pre-defined threshold, is explored in [29]. Neither of these approaches reported the results on real-world VMM or hypervisor products.

Hypervisor scheduling algorithm is also explored. Communication-aware CPU scheduling algorithm for co-located multi-tier applications is explored in [10]. Hybrid scheduling framework for virtual machine systems is explored when the virtual machine is used to execute the concurrent applications [30]. They do not address the virtualization efficiency issue of spin-lock in MP guest.

Memory virtualization technologies have been widely studied in the literature. Barham improves memory virtualization performance with paravirtualization by modifying the guest OS to batch Guest Page Table access operations [6]. Adams analyzed architecture level events of page table updates, context switches, and performance of Shadow Page Table [2]. However, they do not fully take advantage of modern hardware technologies to maximize resource utilization. Waldspurger employed ballooning technique, content-based page sharing and hot I/O page remapping to share memory among guests [25]. Gupta leverages sub-page level sharing (through page patching) and in-core memory compression [11]. However, their impacts on system level scalability are unclear.

Software optimizations to PV I/O virtualization have also been widely studied, such as page remapping and batch packet transferring [9], Xenloop [26], scheduler [14][17], virtual switch enhancement, and transmit queue length optimization [14]. However, none of them can completely eliminate the hypervisor intervention to performance packet movement, and therefore still have the high overheads due to I/O packet copy, latency, and cache pollution.

Hardware-assisted solutions are proposed to PV I/O to achieve high-performance, such as VMDq [22], and self-virtualized devices [20]. In these solutions, each VM is assigned a portion of resources, or a pair of queues, to transmit and receive packets with reduced VMM intervention. SR-IOV [18][8] uses IOMMU for memory protection and address translation. VMM-Bypass I/O is explored in [15], extending OS-bypass design of InfiniBand software stack to bypass VMM for performance. Neither of these approaches analyzed the impacts on system level scalability.

VII. CONCLUSION

Traditional virtualization technologies incur very high virtualization overheads, especially on the modern, multi-core and/or overcommitted systems. For instance, Shadow Page Table suffers from excessive VM-exit due to the write protection on Guest Page Table, and high synchronization overheads due to lock contentions on SPT accesses for the MP guest.

We have implemented and optimized the support of advanced hardware accelerations in the latest version of Xen, including Pause Loop Exit (PLE), Extended Page Table (EPT), and Single Root I/O Virtualization (SR-IOV). The experimental results demonstrate very good performance and scalability on the multi-core and overcommitted system, for both micro-benchmark and a server consolidation benchmark. The results show an up to 77% improvement in the server consolidation benchmark (49% of which due to EPT and another 28% due to SR-IOV), and an up to 14% improvements in the micro-benchmarks due to PLE.

ACKNOWLEDGMENT

The other members contributing to this study include Xiaowei Yang, Xiaohui Xin, Yu Zhao and Intel Linux OS VMM team. We also would like to thank the helpful academic works sponsored by the National Natural Science Foundation of China (Grant No. 60773093, 60970107, 60970108), the Science and Technology Commission of Shanghai Municipality (09510701600, 10511500102, 10DZ1500200).

REFERENCES

[1] Amazon Elastic Compute Cloud User Guide, http://aws.amazon.com/

[2] Keith Adams, Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization, ASPLOS-12, San Jose, CA, 2006, pp. 2-13

[3] Padma Apparao, Ravi Iyer, Xiaomin Zhang, Don Newell, Tom Adelmeyer, Characterization & analysis of a server consolidation benchmark, Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Seattle, WA, 2008, pp. 21-30

[4] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini and Srilatha Manne, Accelerating Two-Dimensional PageWalks for Virtualized Systems, ASPLOS-13, Seattle, WA, 2008, pp. 26-35

[5] A. J. Bernstein, Program Analysis for Parallel Processing, IEEE Trans. on Electronic Computers". EC-15, pp. 757–62, 1966

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, Xen and the art of virtualization, In proceedings of the 19th ACM symposium on Operating Systems Principles, Bolton Landing, NY, 2003, 164-177

[7] J. P. Casazza, M. Greenfield, K. Shi, Redefining Server Performance Characterization for Virtualization Benchmarking, Intel technology Journal, Volume 10, Issue 03

[8] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, Haibing Guan, High Performance Network Virtualization with SR-IOV, HPCA-16, Bangalore, India, 2010.

[9] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williams, Safe hardware access with the Xen virtual machine monitor, In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure, Boston, MA, 2004.

[10] Sriram Govindan, Arjun R. Nath, Amitayu Das, Bhuvan Urgaonkar, Anand Sivasubramaniam, Xen and Co.: Communication-aware CPU Scheduling for Consolidated Xen-based Hosting Platforms, Proceedings of the 3rd ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, San Diego, CA, 2007, pp. 126-136

[11] D..Gupta, etc, Difference Engine: Harnessing Memory Redundancy in Virtual Machines, In Proc. of the 8th OSDI, 2008, Sandiego, CA.

[12] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developers' Manual, http://www.intel.com/products/processor/manuals/index.htm.

[13] Intel Corporation, Intel® 82599 10 Gigabit Ethernet Controller Datasheet. http://www.intel.com

[14] G. Liao, D. Guo, L. Bhuyan, S. R King, Software techniques to improve virtualized I/O performance on multi-core systems. Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, San Jose, CA, 2008, 161-170

[15] J. Liu, et al. High Performance VMM-Bypass I/O in Virtual Machines. Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2006, 3-3

[16] A. Menon, A. L. Cox, W. Zwaenepoel, Optimizing Network Virtualization in Xen. Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2006, 15-28.

[17] D. Ongaro, A. L. Cox and S. Rixner, Scheduling I/O in Virtual Machine Monitors. Proceedings of 4th ACM/USENIX International Conference on Virtual Execution Environments, Seattle, WA, 2008, 1-10.

[18] PCI Special Interest Group, http://www.pcisig.com/home

[19] Gerald J. Popek, Robert P. Goldberg, Formal requirements for virtualizable third generation architectures, Communications of the ACM, v.17 n.7, pp. 412-421, July 1974

[20] H. Raj, K. Schwan, High performance and scalable I/O virtualization via self-virtualized devices. Proceedings of the 16th international symposium on high performance distributed computing, Monterrey, CA, 2007

[21] John Scott Robin and Cynthia E. Irvine, Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, Proc. 9th USENIX Security Symposium, Denver, CO, 2000

[22] J. R. Santos, Y. Turner, G. Janakiraman and I. Pratt, Bridging the Gap between Software and Hardware Techniques for I/O Virtualization, Proceedings of the USENIX Annual Technical Conference, Boston, MA, 2008. 29-42

[23] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, L. Smith, Intel Virtualization Technology, 38(5), Los Alamitos, CA, IEEE Computer Society Press , 2005, 48–56

[24] V. Uhlig, J. Levasseur, E. Skoglund, U. Dannowski, Towards scalable multiprocessor virtual machines, In Virtual Machine Research and Technology Symposium (2004), USENIX, pp. 43–56.

[25] C. A. Waldspurger, Memory resource management in VMware ESX server, ACM SIGOPS Operating Systems Review, v.36

[26] J. Wang, K. Wright, and K. Gopalan, XenLoop: A Transparent High Performance Inter-VM Network Loopback, in Proceedings of the 17th international symposium on High performance distributed computing, Boston, MA, 2008, 109-118

[27] Philip M. Wells, Koushik Chakraborty, Gurindar S. Sohi, Dynamic heterogeneity and the need for multicore virtualization, ACM SIGOPS Operating Systems Review, v.43 n.2, April 2009

[28] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, 2002.

[29] P. M. Wells, K. Chakraborty, G. S. Sohi, Hardware support for spin management in overcommitted virtual machines, In Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (PACT'06) (Seattle, Washington, USA, Sept. 2006), ACM, pp. 124–133.

[30] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu, Hybrid Scheduling Framework for Virtual Machine Systems, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Washington, DC, 2009, pp. 111-120.

[31] K. K. Yue, D. J. Lilja. An Effective Processor Allocation Strategy for Multiprogrammed Shared-Memory Multiprocessors. TPDS. IEEE Transactions on Parallel and Distributed Systems (TPDS), VOL. 8, NO. 12, 1997.

Tackling the Challenges of Server Consolidation on Multi-Core Systems

Hui Lv, Xudong Zheng, Zhiteng Huang, Jiangang Duan
Intel Asia-Pacific Research & Development Ltd.
{hui.lv xudong.zheng zhiteng.huang jianggang.duan}@intel.com

Abstract — With increasing demand to reduce system operation cost amidst growing adoption of virtualization technologies, multiple servers on a single physical chip is fast seeing typical usage in modern data centers. This trend has become more obvious with advances in system design that puts more and more multi-core CPUs into one system. As a result, it is interesting to investigate the challenges of virtualization on top of a multi-core system and the scalability of the consolidation workload on top of it.

With this objective, we conduct a comprehensive analysis of an industry consolidation workload on top of a state of the art Intel system that consists of 64 physical threads. Our findings reveal that careful considerations must be taken in I/O virtualization and VM scheduling to achieve high scalability under the consolidation environment in a multi-core system.

Against this backdrop, we carried out some preliminary work to demonstrate optimization opportunities. In particular, applying interrupt balance method brings in 16% performance boost whereas a finer-grained scheduler prototype added another 6% performance lift. Based on our analyses, existing VMMs should be enhanced with a) a well-designed multi-threaded I/O virtualization implementation and b) a scheduler with finer grained control on priorities.

I. INTRODUCTION

Computer performance has improved greatly in the past couple of decades due to advances in hardware design and manufacturing excellence. With the computing industry entering the multi-core era, more and more cores are being integrated on a single die [1] [2]. This evolution of multi-core systems not only leads to significant usage innovation in the IT industry but also brings about great challenges to software developers as to how to exploit parallelism of their applications to fully utilize the abundant hardware resources. These challenges are even more critical in the virtualization technology (VT) area.

Modern virtual machine monitors (VMMs) or hypervisors (such as VMWare [3], Hyper-V [4] and Xen [5]), which are popular in today's data centers, allow multiple virtual machines (VMs) to share a single hardware platform. To explore the performance potential of hardware resources, consolidating multiple servers onto one physical machine is fast becoming a representative virtualization usage pattern in datacenters. However, the capacity management of VMs is not trivial for a VMM as the resource demands for enterprise applications vary over time, and the problem is even more severe on multi-core systems with many VMs on which various applications are running simultaneously.

In this paper, we investigate and analyze the performance scalability and characterization of vConsolidation [6] [7], a representative consolation workload, on top of an Intel four-socket multi-core system. Our investigation reveals that the single-threaded I/O virtualization is the root cause of low performance for I/O intensive workloads. In addition, management challenges brought about by the complexity of virtualized environments in multi-core systems tend to introduce unfairness of CPU resources sharing between VMs, and this further leads to low CPU efficiency. Last but not least, our analysis also clearly shows that a scheduler with fine-grained resource control could offer better scalability and higher performance.

We also carried out some preliminary work to demonstrate potential optimization opportunities. To resolve the issue caused by single threaded I/O virtualization, we propose a workaround to balance the IRQ across different cores, which results in 16% performance improvement for the vConsolidation workload. We also compare two VMM schedulers and show that the one with fine-grained CPU source control can improve the performance of vConsolidation by another 6%. There is still considerable headroom in terms of consolidated performance in the multi-core system. We urge the need for more decent structured virtualization solutions with a well-designed multi-threaded implementation for I/O virtualization and a scheduler with finer grained control.

The main contribution of this paper is as follows: 1) Compared to previous evaluation of the consolidation workload, our evaluation is the first to evaluate its scalability on top of a state of the art 4-socket multi-core system; 2) We identify I/O virtualization in Xen as a performance bottleneck in multi-core systems, and propose an IRQ balance method that can significantly improve the performance of the consolidation workload; and 3) we show that the current resource scheduler is unfair and inefficient due to dependency between different workloads in consolidation environment,and that a scheduler with finer grained control is required.

The rest of this paper is organized as follows. Section 2 presents the background of server consolidation and a brief

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

introduction of Xen VMM. Section 3 introduces the experimental methodology and the consolidation workload we use. In Section 4, the challenges brought about by server consolidation on multi-core systems are discussed in great detail, followed by the illustration of two approaches to tackle the challenges using micro-benchmarks in Section 5. The performance benefits after optimization is analyzed in Section 6. Section 7 describes related work, and Section 8 concludes this paper.

II. Background

A. Server Consolidation

Multiple virtual machines are allowed to run on the same physical hardware in system virtualization, so as to increase system utilization and reduce cost. Server consolidation reduces the number and variety of components in the environment. This may not be limited to servers but also to other physical elements such as tapes, disks, network devices and connections, operating systems, and peripherals involved in server consolidation. It becomes easy to move and change systems, applications, and peripherals with fewer hardware and software standards to manage. Although the physical hardware is shared across the virtual machines, the virtual machines are fully isolated from each other and each virtual machine runs a separate operating system instance and applications.

Performance characterization for server consolidation is useful for deployment with fair sharing of resources, providing feedback to IT administrators and platform architects, projecting and optimizing future platform performance and so on. vConsolidation developed by Intel and VMmark [9] developed by VMware are two popular proposed benchmarks for virtualization consolidation. In this paper, vConsolidation is adopted for performance analysis of server consolidation. We propose some recommendations in Section IV for resolving virtualization bottlenecks and improving the performance of vConsolidation.

B. XEN Virtual Machine Monitor

A virtual machine monitor [10] [11] [12] [12] allows multiple operating systems to share a single machine safely. It isolates operating systems and controls accesses to hardware resources. As an open source VMM, Xen is widely used and is representative enough. The organization of Xen [5] is depicted in Figure 1. Xen consists of two elements: the hypervisor and the driver domain. The hypervisor provides an abstraction layer between the guest operating systems and the actual hardware. The driver domain, a privileged VM, called domain0, manages other guest VMs, called domainU. One of the major functions of the driver domain is to conduct real I/O operations to a bare device on behalf of domainU to implement a reliable I/O architecture [14] [15] [17] [35]. A driver domain can operate existing device drivers and multiplexing software such as a flexible network bridge. This I/O model requires guest domains to use virtual device drivers for transparent I/O access.

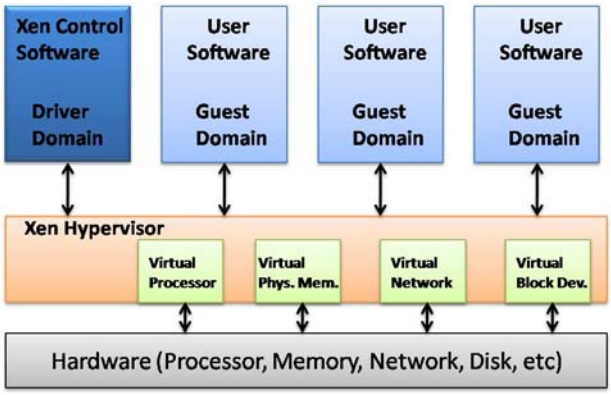


Figure 1: The Xen virtual machine environment

III. Measurement-based Methodology and Workload

A. The vConsolidation Benchmark

vConsolidation, a benchmark developed by Intel, is used to measure the performance of server consolidation. As shown in Figure 2, it simultaneously runs multiple sets of different virtual machines on the same physical server. Each set includes four virtual machines running different workloads of database, web server, java OLTP application and mail server with its own operating system. Besides, there is a fifth idle virtual machine running no workload. These five virtual machines compose a consolidation stack unit (CSU). Depending on the available server resources and server class, one or more CSUs can be added to saturate a given system under test (SUT).

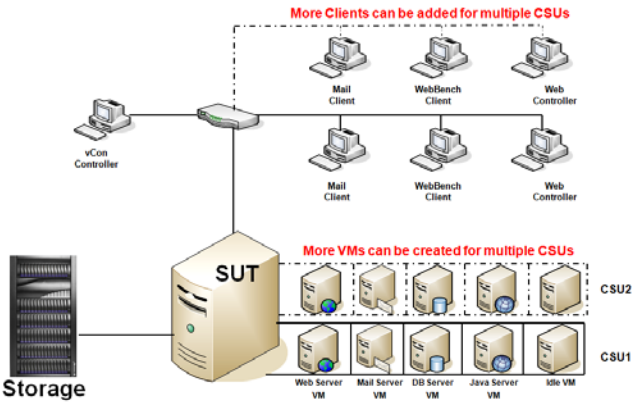


Figure 2: vConsolidation block diagram

Table 1 describes configurations of each virtual machine in a CSU. Several popular benchmarks are modified or customized here to simulate typical IT environment applications, including webbench [28] for web server, loadsim [31] for mail server, specjbb [29] for Java OLTP application and sysbench [32] for database.

Table 1 Virtual machine configurations for each CSU

	vCPUs	vMemory	Operating System	Application
Web Server (webbench)	2	1.5G	Windows 2003 32-bit	IIS
Mail Server (loadsim)	1	1.5G	Windows 2003 32-bit	Exchange
Database (sysbench)	2	1.5G	Windows 2003 64-bit	MS SQL
Java OLTP (specjbb)	2	2.0G	Windows 2003 64-bit	BEA JVM
Idle	1	0.4G	Windows 2003 32-bit	

During the tests, the four workloads report their own throughput as the performance metrics. Table 2 shows the performance of 1CSU as an example. To avoid the range difference of workload throughput, we use this 1CSU number as reference to normalize final performance metrics.

Table 2 1CSU data as the reference number

	webbench	specjbb	Sysbench	loadsim
Perf.	2857	30930	200	17.02

Equations below calculate the workload throughputs ($\boldsymbol{T_{workload}}$) and the system throughput ($\boldsymbol{T_{system}}$). The value of $PerfRef_{workload}$ refers to the numbers in Table 2 for each workload.

$$T_{workload} = \sum_{i=1}^{CSU\_num} \left(\frac{Perf_{workload}[i]}{Perf\ \mathrm{Re}\ f_{workload}} \right)$$

$$T_{system} = \frac{T_{webbench} + T_{specjbb} + T_{sysbench} + T_{loadsim}}{4}$$

B. Measurement Platform

The server under test is the latest Intel Xeon-based server with four 2.26GHz processors. Each processor has 8 cores, 16 threads with Hyper-Threading technology [18] on and 24MB shared L3 cache. Xen 3.4.0 is used as the virtual machine monitor. We allocate enough storage and network devices to make sure that there is no hardware bottleneck. One LSI HBA is used to connect to an external disk enclosure. Each CSU is assigned 2x64GB Intel solid status disk as storage. One more Intel Gbit NIC will be added into the system every 6 CSU.

IV. Challenges For VT in Server Consolidation

Considerable amount of work has been done in recent years to reduce virtualization overheads and boost performance:

1) EPT [19] provides a hardware assistant to accelerate virtualization memory translation instead of software management to reduce overhead.

2) Intel Virtualization Technology for Directed I/O (Intel VT-d) [21] helps to obtain direct access to I/O resource without requiring the VMM to provide emulated device drivers.

3) PCI-SIG also supplements the Single Root I/O Virtualization and Sharing (SR-IOV) [22] [34] which enables efficient sharing of a single I/O device among multiple VMs. Coupled with VT-d, SR-IOV provides a foundation for efficiently utilizing I/O resources as reaching near-native I/O performance. However, all of these are hardware solutions. A more flexible and widely used software solution to improve I/O performance is Para-virt (PV) driver.

4) PV driver bypasses I/O emulation, in an attempt to reduce heavy overhead brought by software emulation layer. From then, present virtualization is "good enough" to reach close-native performance.

However, as the number of cores is growing, tremendous VMs accommodating various kinds of workloads can be centralized in terms of sever consolidation to share the uniform resources in the multi-core system. The capacity management of VMs becomes harder and harder for VMM due to complex environment [23]. Through investigation and analysis of scalability of the performance of the consolidation workload with the mainstream virtualization technologies, we identify the following two key challenges. Although this work is based on the Xen hypervisor, we believe the challenges proposed are applicable to other VMMs.

A. IO Virtualization Challenges

To provide insights into behaviors of server consolidation in multi-core systems with virtualization technologies, we present the thread scalability performance of vConsolidation in the latest 4-socket system. The number of cores/threads varies from 4 cores/8 threads to 32 cores/64 threads through enabling 1, 2 and 4 cores in each processor and with the help of Hyper-Threading technology that enables two simultaneous threads in unison on one core.

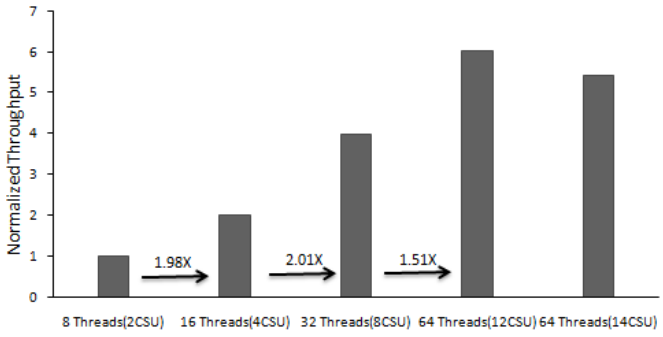


Figure 3: vConsolidation thread scaling results

As shown in Figure 3, when the number of cores in the system is small, the overall system throughput of vConsolidation scales up linearly with the core numbers. The

16-thread system provides 98 percent increase in throughput over that with 8 threads and the 32-thread system achieves around 100 percent throughput increase over that with 32 threads. These indicate the performance advantage of multi-core processors. However, in a 64-thread system, the peak performance is reached at 12 CSU, providing 51 percent more throughput than a 32-thread system. This scaling ratio is much lower than systems with fewer threads.

Turning our attention to the cause of this low scaling ratio from 32 to 64 threads, the behaviors of each component in vConsolidation between 32-thread (8CSU) and 64-thread (12CSU, 14CSU) are compared in Figure 4. For peak performance comparison: with the help of the EPT feature, the memory related workload, sysbench, exhibits excellent scaling ratio of 2.20x from 32 to 64 threads. Meanwhile, the CPU intensive workload, specjbb, also scales well – the ratio is 1.73x – indicating that VMM can handle CPU intensive workloads as well. However, the throughput of webbench (network I/O intensive) is flat and the throughput of loadsim (disk I/O intensive) improves slightly with a 1.32x scaling ratio from 32 to 64 threads. This implies that some I/O related bottleneck may be happening in a 64-thread system. As shown in Table 3, data from the 'xenoprofile' tool [33] shows that the CPU utilization of function 'netbk' (the backend interface of network) decreases in a 64-thread system. This evidently shows that the network backend is not able to function efficiently under this condition and the overall network throughput does not scale.

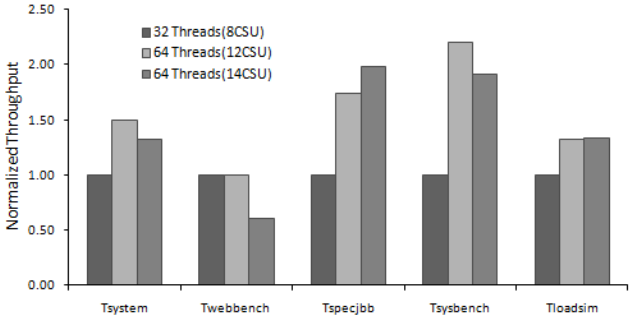


Figure 4: vConsolidation components core scaling from 32 to 64 threads

Table 3. CPU utilization of function 'netbk'

	8 Threads	16 Threads	32 Threads	64 Threads
netbk	**3.04%**	**4.79%**	**9.02%**	**5.36%**

Altering to a 64-thread system, as illustrated in Figure 5, the dotted line shows the varying throughput of webbench with increasing CSU stresses. It is observed that total throughput of webbench starts to decrease sharply after 8CSU. The reason can be found in Figure 6: in the current I/O structure, most interrupts are simply delivered to VCPU0 in the driver domain by default. This problem does not exist when the I/O stress in the system is generally low. However, with the increasing capability of multi-core systems to host more VMs containing I/O intensive applications, multiple interrupts will overwhelm VCPU0, while other VCPUs will remain relatively idle. This IRQ delivery approach therefore becomes a major bottleneck for I/O scalability.

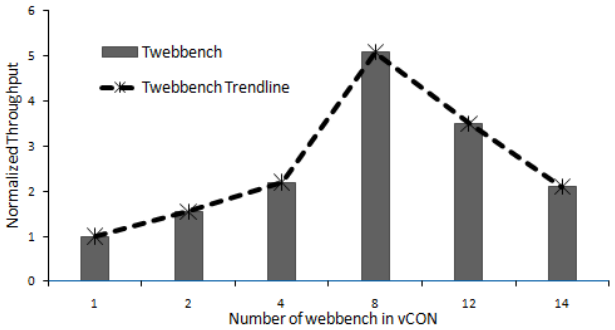


Figure 5: Performance trend of webbench in a 64-thread system

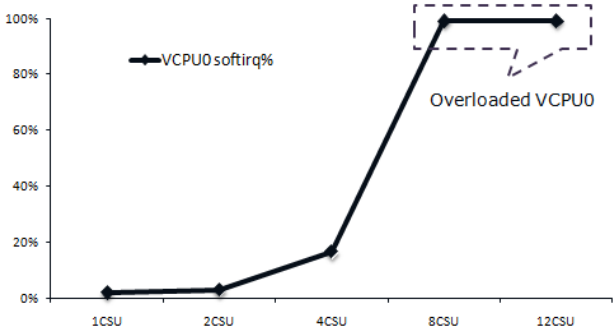


Figure 6: VCPU0 is overloaded by increasing I/O stress

Moreover, the implementation of the current I/O subsystem in Xen did not have multi-core in mind [14]. The I/O backend driver has only two contexts for deferred kernel tasks for processing data – one single transmit context and one single receive context. It easily becomes another bottleneck for the I/O scalability when I/O stresses increase dramatically. In this paper, to solve the VCPU0 overcommitted issue, we have studied the relationship between virtual devices and physical devices (like VNIF/NIC and VBD/disk) and propose the IRQ balance way in Section A, Part V.

B. Resource Management Challenges

The usage of virtualization in server consolidation is progressively accommodating diverse and unpredictable workloads. These unpredictable workloads make CPU resources allocation difficult [25]. Since VMM lacks knowledge about existing tasks in each guest operating system, it has difficulty in considering mixed workloads in VMs. Such a semantic gap would degrade performance when many VMs are consolidated and their workloads are diversified. For example, the default scheduler in Xen, credit scheduler, is coarse-grained and does not take correlation and dependency of different workloads into account, which introduces unfair and low efficient allocation of CPU resource, especially for short response workloads. In an attempt to solve the problem of performance bias for short response latency workloads, the credit scheduler has already added the additional state BOOST in addition to OVER and UNDER [15] [17], but there are also

many other aspects in the current scheduler, which make it unfair at scheduling latency-sensitive VMs, including long time slice (30ms), sorting by priority rather than credit, and the probabilistic debiting of credit 10ms at a time.

Fairness is the ability of a VM to get its "fair share" of CPU. The unfairness issue can be easily observed in the vConsolidation experiment as depicted in Figure 7. In the 64-thread system, with CSU number growing, the CPU utilization for each workload should increase proportionally. However, it is not the case for webbench and sysbench, when the number of VMs reaches certain degree (from 12CSU to 14CSU). The CPU intensive workload, specjbb, is likely to occupy more CPU resource, while webbench and sysbench are likely to occupy less CPU resource. Therefore, for the aspect of CPU resources allocation, the current scheduler is biased due to the simple VCPU-level (virtual CPU is the schedule unit) scheduling algorithm in the credit scheduler.

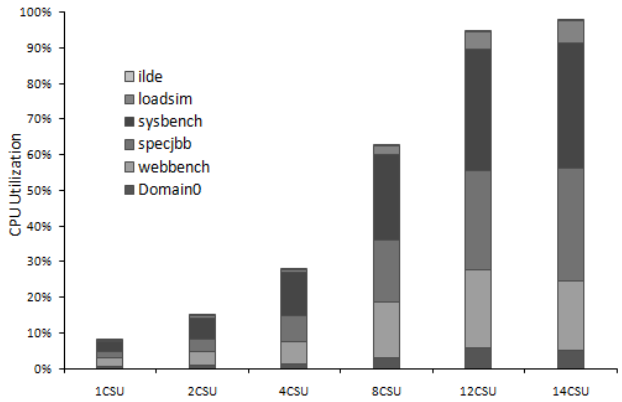


Figure 7: CPU efficiency for individual workloads in vConsolidation

The complexity of the virtualized environments in server consolidation brings the resource management challenge for VMM. Results in Figure 7 show that the current resource scheduler is unfair and less efficient. As a concrete example of different types of resource management involved, we will compare and analyze the default CPU scheduler in Xen with a finer-grained scheduler in Section B, Part V, so as to show that cost effective management methods should be applied to guarantee workloads a fair and efficient sharing of uniform resources.

V. Implementations of Proposed Optimization Methods

It would make sense to deliver all I/O interrupts to VCPU0 provided with the single-threaded tasklet in the driver domain. However it limits the I/O backend's ability to utilize additional computing resources. To resolve this issue, an IRQ balance way inspired from native Linux implementation to release the pressure on VCPU0 is proposed in this paper. Moreover, concerning the approach to probabilistic debiting of credit in the default credit scheduler, we examine a scheduler with accurate credits debiting method [30] to show its performance benefits in this section.

A. IRQ Balance Delivery Method

The communication between guest domain and real physical devices is implemented by the frontend in the guest domain and the backend in the driver domain. A virtual frontend driver in the guest domain communicates with a corresponding virtual backend driver, which resides in the driver domain and forwards delivered I/O requests to a native device driver. A frontend driver and a backend driver notify each other of an I/O event through an event channel. The event channel mechanism virtualizes a hardware interrupt. A virtual interrupt is pending in the corresponding event channel and then is delivered into the target domain when the domain is scheduled. The latency between pending and delivered events obviously depends on the underlying VM scheduling mechanism. Figure 8 shows the network I/O architecture in Xen.

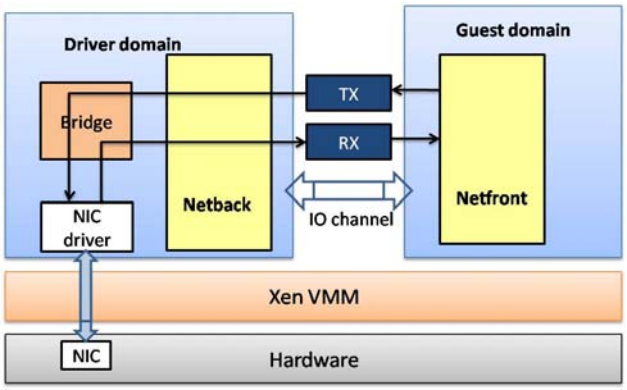


Figure 8: PV driver network I/O architecture in Xen

Xen adopts split-driver I/O architecture as discussed above. In this architecture, domain0, as the driver domain, plays a key role in the communication between VMs and the real devices to avoid bottlenecks to reduce overheads and achieve high I/O performance. However, in current Xen, most of the interrupts (e.g. disk, VBD, NIC, VNIF) are directly delivered to VCPU0 in domain0 by default. With the increasing I/O stress in the powerful multi-core system, VCPU0 is overloaded as Figure 6 demonstrates, while other VCPUs remain idle, which leads to performance degradation for the I/O intensive workload.

In native Linux [26], the SMP IRQ Affinity bitmask /proc/irq/nnn/smp_affinity can be set to designate which CPUs are permitted to process specific interrupts. A daemon called *irqbalance* is used to dynamically distribute IRQs across processors. If enabled, it iteratively alters the *smp_affinity* bitmasks to perform the balancing. The balance mechanism should also be imported into domain0, the modified Linux, to release the heavy pressure on VCPU0. However, several things must be taken into account for implementing the balance mechanism in domain0 considering the interactional interrupts of NIC/VNIF and disk/VBD

While studying the performance behaviors of network intensive workload by selecting different interrupts delivery

mechanisms, a simplified consolidation environment is set in a 32-thread system with 16 network intensive workloads, webbench, running in 2-VCPU virtual machine respectively. Three 1Gb NICs are employed to avoid network bandwidth limitation. Different ways to assign interrupts are tested in Figure 9. In this experiment, Method A, as the default way in Xen, delivers interrupts to VCPU0 so as to make VCPU0 overloaded while other VCPUs are idle. In order to keep balance, IRQs are distributed to various VCPUs in domian0 to avoid one of the VCPUs overloaded in Method B. In Method C, the relationship between the physical NIC and its corresponding VNIFs is considered and the interrupts from each physical NIC and its corresponding VNIFs are assigned to the same VCPU. In this way, the related interrupts are processed in the same VCPU so as to reduce the overhead of migrations between different VCPUs. The experiment results illustrate that Method C has about 9% performance advantage compared to default Method A and 1.3% compared to Method B. Without considering the relationship between the physical NIC and corresponding VNIFs, Method B shows lower CPU efficiency than Method C. We therefore recommend Method C for keeping IRQ balance for the driver domain for future I/O virtualization designs.

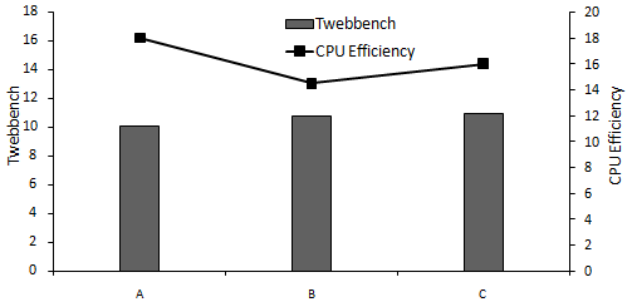


Figure 9: Throughputs comparison between different ways to assign interrupts in domain0

B. Scheduler Evaluation in Multi-Core System

The VMM functions as an abstraction layer of the real physical devices. As a result, scheduling in virtualization is based on virtual CPUs, because physical CPUs are transparent to the guest VMs. Three different CPU schedulers were introduced in recent years – Borrowed Virtual Time (BVT) [27], Simple Earliest Deadline First (SEDF) [8] and Credit Scheduler – all of which allow the user to specify CPU allocation via CPU share (weights).

The credit scheduler, Xen's latest scheduler, is a proportional share scheduler with a load balancing feature for SMP systems. The virtue of the credit scheduler is the simplicity of the operation with reasonable fairness guarantee and performance. The credits are debited on periodic scheduler interrupts so that the scheduler is fairly sharing the processor resources by approximation (probabilistic). Because the credits are updated every 10 ms, the domain running for less than 10ms will not have any debits debited at all. In the condition when all domains are primarily using processor resources, it is unlikely to have a significant impact on the fair sharing of the processor resources. However, when short response latency work is assigned –a networking application for example –this policy may be ineffective.

Rather than debiting a full 10ms of credits in a scheduler tick (probabilistic), the finer-grained scheduler debits credits accurately based on time stamps [30]. Besides every scheduler tick, credits are also updated when scheduler happens to avoid the previous VM stealing the CPU utilization of the next one as shown in Figure 10.

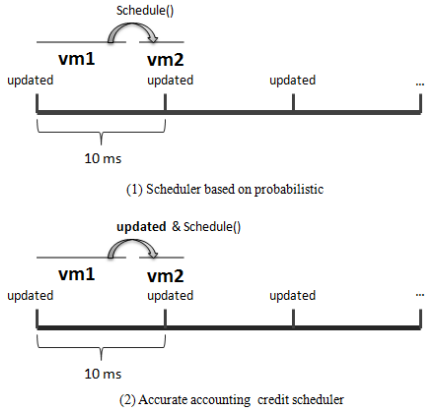


Figure 10: (1) Scheduler based on probability (2) Accurate accounting credit scheduler

Table 4. Function implementation of fine-grained scheduler

csched_schedule

{
0. credit updated
1. Select next runnable local VCPU
2. SMP Load Balancing
3. Return ret (struct task_slice)
}

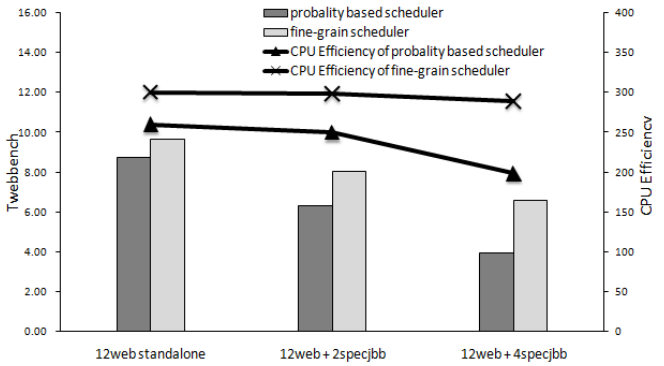


Figure 11: Webbench performance comparison between two schedulers with increasing stress by adding more VMs containing CPU intensive workload

A *start_time* is initialized for each scheduled VM and tracked when updating credits. During every time tick or the process of schedule, the time period '*now – start_time'* responds to credits thus the current domain is debited by converting the time period to relative credits instead of fixed credits, leading to the

accurate credit calculation. After debiting the credits, the value of start_time is replaced by now. Table 4 summarizes the current function of the accurate scheduler, which shows that only 'step 0' is added into the new scheduler. The added overhead is small leading to a big performance gain for the accurate scheduler.

To study the performance behaviors of network intensive workload by selecting different schedulers, we setup a simplified consolidated environment within a 16-core system. 12 network intensive workloads, webbench, run in 2-VCPU VM respectively. In order to increase the stress of the system, 2-VCPU VMs containing CPU intensive workloads, specjbb, are added. As shown in Figure 11, the line of probability based scheduler shows that with increasing stress in the system, the CPU efficiency of overall webbench drops greatly, which implies that the scheduler is lowly efficient to the network intensive workload. After applying the accurate credit debiting scheduler, the corresponding line shows that with increasing stress in the 16-core system, the CPU efficiency of overall webbench is flat, implying that the accurate scheduler is efficient. Moreover, the overall throughput with the accurate accounting scheduler is 61% higher than that of the probability based scheduler.

VI. Results and Analysis

A. Experimental Results after Optimizations

By adopting the IRQ balance method, two more CSU can be added to saturate a 64-thread system. The system throughput improves 16% as shown in Figure 12. The I/O intensive workloads in vConsolidation, webbench and loadsim, contribute most for the overall system throughput improvement after distributing interrupts from VCPU0 to VCPU4 as shown in Table 5. The throughput of webbench and loadsim increase 47% and 30% respectively, as depicted in Figure 13 in the condition of a 64-thread system with 14CSU stress.

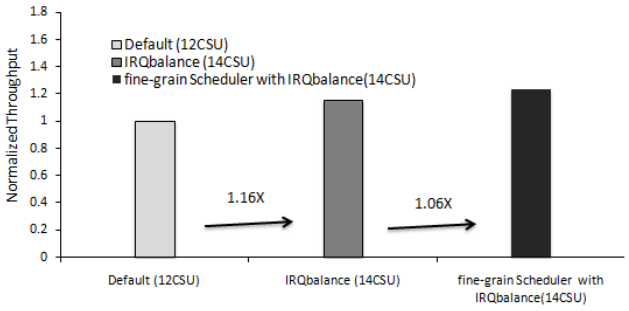


Figure 12: The methods of IRQbalance and fine-grain scheduler improve the overall performance

The ideal trend line (without scalability issue) from curve fitting is drawn in Figure 14 to predict the performance of webbench. It is observed that there is still a big gap for webbench to achieve this expected throughput. After applying the IRQ balance method, the throughput of webbench does not either reduce, or increase beyond 8CSU. (The bandwidth of the physical NIC is sufficient and it is not the bottleneck). The single-threaded tasklet in the backend cannot serve more requests when I/O stresses increase to a certain degree. Therefore, it is expected that the performance of I/O in current virtualization will not benefit from future multi-core platforms with many more cores due to the single-threaded limitation. Although, a multi-threaded solution is proposed in [9] and the multi-threaded backend for Linux VM has been developed recently, they are not ready for high I/O performance usage. Therefore, to solve this scalability issue completely, a thoughtful multi-threaded I/O virtualization solution is urgently needed.

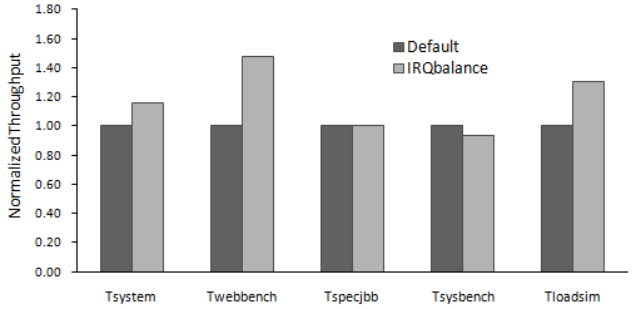


Figure 13: IRQbalance way improves I/O performance

Table 5. CPU utilization of VCPUs

	VCPU0	VCPU1	VCPU2	VCPU3
Average softirq%	23.26%	44.39%	44.02%	34.84%

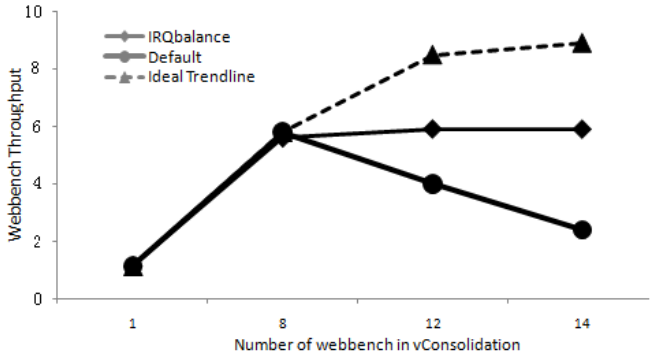


Figure 14: Comparisons of webbench throughput trends

By adopting the proposed finer-grained scheduler, the system throughput increases another 6% as shown in Figure 12. We select CPU efficiency, or performance per CPU utilization, as a metric. Higher CPU efficiency implies higher performance achieveable by one CPU unit. To present the influence of CPU efficiency from these two schedulers for a multi-core system, we conduct a comparison between an 8-thread and a 64-thread system. As shown in Figure 15, after applying the finer-grained scheduler, the CPU efficiency of all components in vConsolidation is flat in the 8-thread system, while it improves 6%, 8%, 8% and 6% for webbench, specjbb, sysbench and loadsim respectively in the 64-thread system in Figure 16. These results demonstrate that the default scheduler has low

efficiency in a multi-core system and that a finer scheduler, with accurate credits debiting method, can benefit CPU efficiency so as to improve the overall system performance.

Although the current VCPU-level scheduling mechanism is simple and supports fairness to some extent, it has limitations due to the semantic gap. The lack of knowledge about the heterogeneous workloads could lead to performance degradation. To guarantee the fairness and efficiency of CPU resource allocation between virtual machines, it is urgent to define a finer level scheduler. For example, a guest-level scheduler with knowledge of its internal workload could give a more effective resource allocation whereas a scheduler sorting VCPUs by remaining credits could support finer grained control on priorities.

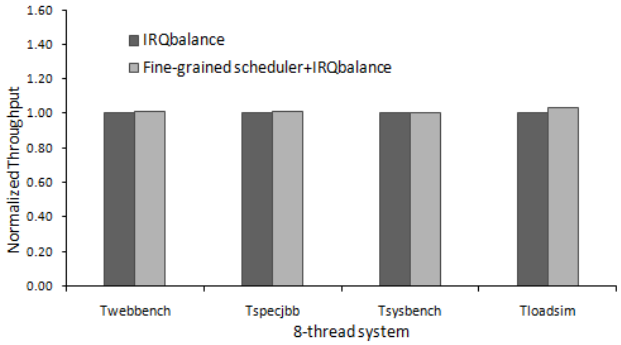


Figure 15: CPU efficiency is flat after applying a fine-grain scheduler in an 8-thread system

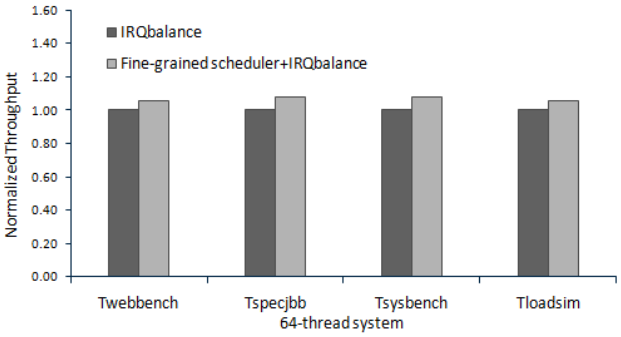


Figure 16: CPU efficiency improves a lot after applying a fine-grain scheduler in a 64-thread system

B. Performance Prediction

Integrated with the two methods proposed in this paper, the overall system throughput of vConsolidation improves 23%. The scaling ratio grows up to 1.84 going from 32 to 64 threads. We project the performance from curve fitting based on the current core scaling results in Figure 17. Although the current scaling ratio is acceptable from 32 to 64 threads, the predicted scaling ratio is only 1.62 from 64 to 128 threads, indicating that existing bottlenecks, such as single-threaded I/O backend and low efficient scheduler, will degrade the performance of consolidation workload in future multi-core systems. There is still considerable headroom in terms of consolidated performance in the multi-core system. Better structured virtualization solutions are needed to achieve better scalability for future multi-core platforms.

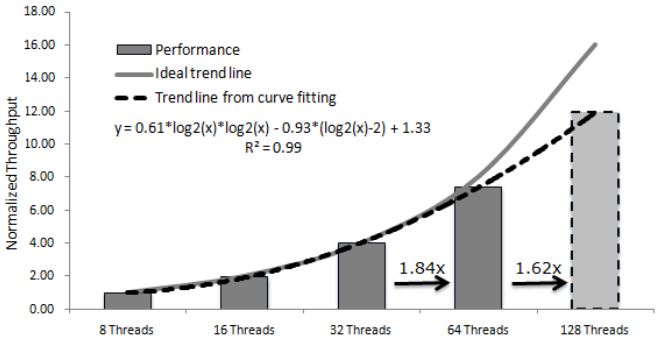


Figure 17: Performance prediction for future multi-core platforms

VII. RELATED WORK

Analyzing and improving virtual consolidation performance is a popular topic in the research community. Studies for more efficient IO architecture [12] [12] [14] [15] [16] and better hypervisor resource scheduler [17] [23] [27] are covered in several previous works.

Menon et al. [12] evaluated network virtualization overheads in the Xen environment using different workloads and under different Xen configurations. Their study focused on networking applications in uni- and multi-processor systems and showed that vitalized network I/O devices are relatively expensive. In [15], Menon et al. presented three ways to optimize the network I/O including redefining the virtual network interfaces of guest domains; optimizing the implementation of the data transfer path; and providing support for guest operating systems to effectively utilize advanced virtual memory features.

Using Xen, Cherkasova et al. [23] studied the impact that three different schedulers have on the throughput of three I/O-intensive benchmarks. In addition to the Credit and SEDF schedulers [8], their study included Xen's BVT [27] scheduler. They evaluate how a single instance of these applications is affected by the choice of scheduling policy. In effect, they evaluate how the scheduler divides the processing resources between the guest domain and the driver domain. Ongaro et al. [17] explored the relationship between domain scheduling in a VMM and I/O performance using multiple guest domains concurrently running different types of applications. In addition, their work examined a number of new and existing extensions to Xen's Credit schedule targeted at improving I/O performance. Finally, their study has shown that latency-sensitive applications will perform best if they are not combined in the same domain with a computing intensive application, but instead are placed within their own domain.

Compared to previous research work leveraging simple workload and hardware, we conducted our measurements and analysis with the state of art hardware. The workload we used is also modified through complex industry benchmark, thus representative enough of real virtual consolidation environments. We believe our performance data and conclusion

are more realistic and can be used as a reference for real IT product system design.

VIII. Conclusions and Future Work

In this paper, through analyzing performance scalability of a representative consolidation workload on the latest four-way X86 multiple core architecture, we identify two areas that present both challenges and opportunities for virtualization in server consolidation: i) the I/O subsystem in today's hypervisor is still a bottleneck, and ii) that the current scheduler can be improved with a finer control algorithm. Regarding these two areas, we proposed potential optimization opportunities: applying an interrupt balance method brings in 16% performance boost and a finer-grained scheduler prototype results in another 6% improvement.

Specifically, this study has shown that the single threaded tasklet in the I/O backend has become the performance bottleneck in the latest multi-core system. Although the IRQ balance way can prevent I/O interrupts from making one VCPU overloaded, the overall throughputs of I/O do not improve. In future work, a multi-threaded I/O structure should be developed to break this bottleneck to improve I/O performance. This study has also shown that VMM schedulers do not achieve the same level of fairness for some compute intensive workloads among consolidation workloads. Although the scheduler prototype with accurate credit debiting can improve the overall system CPU efficiency, there is still much room for future improvement. A guest-level scheduler considering its internal workload and with finer control on priorities is arguably better in allocating the hardware resources well between virtual machines to achieve a better overall system level performance.

References

[1] Intel Corporation. Terascale computing. http://www.intel.com/research/platform/terascale/index.htm.

[2] Intel Corporation. Intel Develops Tera-Scale Research Chips. http://www.intel.com/pressroom/archive/releases/20060926corp_b.htm.

[3] VMware. http://www.vmware.com/.

[4] Microsoft Virtual Server, http://www.microsoft.com/hyper-v-server/

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct. 2003

[6] P. ApparaoNewell, Towards Modeling & Analysis of Consolidated CMP Servers, Workshop on the Design, Analysis, and Simulation of Chip Multi-Processors (dasCMP), 2007

[7] Jeffrey P. Casazza, Redefining server performance characterization for virtualization benchmarking, Intel Technology journal August 2006

[8] I. M. Leslie, D. Mcauley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. IEEE Journal of Selected Areas in Communications, 1996

[9] VMmark http://www.vmware.com/products/vmmark/

[10] M. Rosenblum. VMware's Virtual Platform: A virtual machine monitor for commodity PCs. In Hot Chips 11: Stanford University, Stanford, CA, August 15–17, 1999

[11] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. War_eld, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS), Oct 2004

[12] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In Proceedings of the First ACM/USENIX International Conference on Virtual Execution Environments (VEE), pages 13–23, June 2005

[13] Selvamuthukumar Senthilvelan and Murugappan Senthilvelan. Study of content-based sharing on the xen virtual machine monitor http://www.cs.wisc.edu/~remzi/Classes/736/Spring2005/Projects/Muru-Selva/cs736-report.pdf.

[14] Wiegert, J., et al.: Challenges for Scalable Networking in a Virtualized Server. In: 16th International Conference on Computer Communications and Networks

[15] A. Menon, A.L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In Proceedings of the 2006 USENIX Annual Technical Conference, pages 15–28, June 2006

[16] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *USENIX Annual Technical Conference*, Apr. 2005

[17] Diego Ongaro , Alan L. Cox , Scott Rixner, Scheduling I/O in virtual machine monitors, Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, March 05-07, 2008, Seattle, WA, USA

[18] Susan J. Eggers , Joel S. Emer , Henry M. Levy , Jack L. Lo , Rebecca L. Stamm , Dean M. Tullsen, Simultaneous Multithreading: A Platform for Next-Generation Processors, IEEE Micro, v.17 n.5, p.12-19, September 1997

[19] Xudong Zheng, Jiangang Duan, Shameem F Akhter, Zhidong Yu, Hui Lv, A Consolidation Workload Characterization Study on Modern Platform, CMG'09

[20] Intel Virtualization Technology http://www.intel.com/technology/itj/2006/v10i3/1-hardware/6-vt-x-vt-isolutions.htm

[21] Intel Corporation: Intel Virtualization Technology for Directed I/O. http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art02.pdf

[22] PCI-SIG Single-Root I/O Virtualization and Sharing Specification:www.pcisig.com/specifications/iov/review_zone

[23] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the Three CPU Schedulers in Xen. ACM SIGMETRICS Performance Evaluation Review, 35(2):42–51, 2007

[24] Credit Scheduler. http://wiki.xensource.com/xenwiki/CreditScheduler

[25] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 126–136, New York, NY, USA, 2007. ACM Press

[26] IRQ balance http://irqbalance.org/

[27] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a

general-purpose scheduler. In Proceedings of the 17th ACM SOSP, 1999.
[28] PC Magazine. WebBench 5.0. http://www.pcmag.com/benchmarks/
[29] SPECjbb http://www.spec.org/jbb2005/
[30] Scheduler with accurate credit debiting http://www.mailinglistarchive.com/html/xen-devel@lists.xensource.com/2009-08/msg00911.html
[31] Exchange Server 2003 MAPI Messaging Benchmark 3 http://technet.microsoft.com/en-us/library/cc164328(EXCHG.65).aspx
[32] Sysbench http://sysbench.sourceforge.net/
[33] Xenoprofile http://www.xen.org/files/summit_3/xenoprof_tutorial.pdf
[34] Santos, J.R., Turner, Y., Janakiraman, G., Pratt, I.A.: Bridging the gap between software and hardware techniques for I/O Virtualization. In: ATC'08, Berkeley, CA, USA, USENIX
[35] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In VEE, 2008.

Performance Variations of Two Open-Source Cloud Platforms

Yohei Ueda and Toshio Nakatani

Abstract—**The performance of workloads running on cloud platforms varies significantly depending on the cloud platform configurations. We evaluated the performance variations using two open-source cloud platforms, OpenNebula and Eucalyptus.**

To assess the performance variations on the cloud platforms, we created a representative workload from Wikipedia software and data. The performance with this workload was quite sensitive to two key configuration choices, the physical location of the virtual machine disk images (local disk or NFS), and eager or lazy allocation of the virtual machine disk images. Our performance metrics included (1) the provisioning times for the virtual machines, (2) the elapsed times for two types of batch processing, and (3) the throughputs of two types of Web transactions. The local-disk configuration was 75% slower for provisioning, 2.9 times faster for batch possessing, and 50% faster for Web transactions compared to the NFS configuration. Relative to lazy-allocation, eager-allocation took 2.7 times longer for provisioning and was 43% faster for batch processing, but was only 1.5% faster for Web transactions. Our results indicate that no configuration offers the best performance for all three of the metrics at the same time. If batch processing is more important than provisioning, the local-disk configuration with eager disk allocation should be used. Otherwise, local-disk allocation with lazy allocation should be used.

We also evaluated a multi-tenancy scenario using the Apache DayTrader benchmark on Eucalyptus. The results show that VM provisioning significantly affected the throughputs of DayTrader due to the lack of any disk I/O throttling mechanism.

I. INTRODUCTION

The market demand for cloud computing is shown by the success of Amazon's Elastic Compute Cloud (EC2) [8], and the demands for cloud computing infrastructure are increasing in many enterprises and organizations. There are already competing open-source clouds such as OpenNebula [7], Eucalyptus [5], Nimbus [9], and VCL [10][11].

Although we can easily build cloud environments using such open-source platforms, it is difficult to insure the quality of service, which makes it hard to craft the appropriate service-level agreements for users. The performance of workloads on cloud platforms varies significantly depending on the cloud platform configurations and the concurrent workloads.

In this paper, we evaluated the performance variation of workloads using the two open-source cloud platforms OpenNebula and Eucalyptus. We created a representative workload based on Wikipedia [14], using MediaWiki [12], PHP, MySQL, Lucene [13], Java, and Linux. As data, we used articles from the English version of Wikipedia. For each cloud platform, we provisioned 10 VMs where the software and data were installed, ran and measured batch processing tasks and Web transactions, and then analyzed the performance.

Using the Wikipedia workload, we measured the performance of the cloud platforms for three metrics: (1) the provisioning times for VMs, (2) the elapsed times for batch processing, and (3) the throughputs for two kinds of Web transactions. Our analysis of the results showed that two configuration choices on the cloud platforms significantly affected the performance.

The first configuration choice is the physical location of the VM disk images, on the local disk of the host machine on which the VM runs or on the disk of a central server accessible via NFS (Network File System). Eucalyptus uses the local-disk configuration, whereas OpenNebula supports both the local-disk and NFS configurations.

The second choice is eager or lazy allocation of the VM disk images. With eager disk allocation, all of the physical disk areas for a VM are allocated when it is provisioned. With lazy disk allocation, the physical disk areas for a VM are not allocated when the VM is provisioned, but they are allocated as needed when the guest operating system on the VM first tries to write data. Lazy disk allocation is implemented using *sparse files* [18], in which the operating system does not allocate physical disk areas for file ranges and handles them as holes. Eucalyptus supports only eager disk allocation, whereas OpenNebula supports both eager and lazy disk allocation.

Our experimental results indicate that no configuration offers the best performance for all three of the metrics at the same time. In general, the performance of Web transactions is most important, but it is up to the user whether provisioning time or batch processing is more important. If batch processing is more important than provisioning, the local-disk configuration with eager disk allocation should be used. If provisioning is more important than batch processing, the local-disk allocation with lazy allocation should be used. The NFS configuration is suitable only when provisioning time is more important than the other metrics.

We also measured the interference between provisioning activities and user activities. We created a multi-tenancy workload using Apache Geronimo's DayTrader Benchmark, and the result showed that VM provisioning significantly affected the throughputs of DayTrader due to the lack of a disk I/O capping mechanism, and this caused large performance fluctuations.

The rest of this paper is organized as follows: In Section II, we review related work. In Section III, we describe the architectural design of our target cloud platforms. In Section

Y. Ueda and T. Nakatani are with IBM Research – Tokyo, Yamato-shi, Kanagawa 242-8502 Japan (e-mail: {yohei, nakatani}@jp.ibm.com).

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

IV, the workloads we used to measure performance on our target cloud platforms are described. We present the performance results and analysis in Section V. We discuss possible future work in Section VI, and conclude the paper in Section VII.

II. RELATED WORK

Binnig et al. [4] discussed the difficulties of benchmarking cloud services and the specific problems of applying existing benchmarks to cloud computing. The goal of traditional benchmarks such as TPC-C is to evaluate the average performance of a system using a particular workload with a fixed configuration of software and hardware components. In cloud computing, the configuration may change on demand, so the existing benchmarks are not suitable for capturing the dynamic scalability of cloud computing. They also presented their ideas on how to create better benchmarks for cloud computing with new statistical metrics to assess dynamic scalability. The metrics measure how quickly and smoothly the target cloud service reacts to changes in demand. Their method can compare different cloud services, and is applicable to our target cloud platforms. However, no benchmark implementation based on their method has yet been published, and our target cloud platforms were not covered in their work.

CloudStone [3] is another benchmark for cloud computing. This is a three-tier Web 2.0 application to asses the performance of cloud services. They provide a set of multi-language application code that runs on different language runtimes such as PHP and Ruby and measured the performance on Amazon EC2. They proposed measuring dollars per user per month for benchmarking cloud services. This metric is not an absolute performance value, but a value relative to the charged prices that can be used for comparing different cloud services. Their focus was mainly above the infrastructure level and not on the internal designs of the infrastructures and, once again, our target cloud platforms were not included.

III. DESIGN COMPARISION OF TARGET CLOUD PLATFORMS

In this section, we describe the two open-source cloud platforms in detail.

A. OpenNebula

OpenNebula provides a flexible tool to build a private cloud system on an existing server environment. We used OpenNebula 1.4 for our evaluation.

OpenNebula supports three hypervisors, Xen [1], the Kernel-based Virtual Machine (KVM) [15], and VMware. We chose Xen as the standard hypervisor for comparisons, because Eucalyptus and Amazon EC2 also support Xen.

OpenNebula is highly customizable, so we can build various cloud systems using OpenNebula. However, our intention was to evaluate the cloud platform itself, so we used the default settings whenever possible.

In OpenNebula, there are two types of physical machines. One is a *front end*, and the other is a *cluster node* or just a *node*. A front end accepts requests from cloud end-users, and provides cloud services for them. End users use a command line interface or an XML-RPC API to access the front end. A cluster node hosts VMs using a hypervisor. The VMs on a node are under the control of a front end. The front end periodically checks the status of its VMs and the available resources on each node, and creates or destroys VMs based on the end-user requests. The cloud administrators can customize the placement policies of the VMs on multiple cluster nodes such as the packing, striping, or load-awareness policies. Management between the front end and a node is done using standard SSH, and the cloud administrators do not need to install any agent program on each node.

OpenNebula does not directly support provisioning multiple VMs from a single image. Therefore, when we launch many VMs from a single image, we need to invoke the create command multiple times. This means that there is no optimization for this typical situation.

OpenNebula supports two types of deployment, the *NFS* and *LocalDisk* configurations. In the NFS configuration, the deployed disk images are stored on a central front end machine, and each VM on a node accesses them through NFS. Figure 1 depicts the NFS configuration. In the LocalDisk configuration, the deployed disk images are transferred through SSH connections and copied into a local disk at each node, so each VM on the node can access them without network access. Figure 2 depicts a LocalDisk configuration.

1) OpenNebula NFS configuration

Here are the steps to provision a new VM in an OpenNebula NFS configuration:

1. The front end selects a target node based on the user's placement policy.
2. A root disk image, which contains an operating system and possibly user data and applications, is copied from the front end image repository into an NFS-exported area on the front end. In this step, the copying is only between local disks on the front end.
3. Optionally, empty ephemeral disk images are created and stored into an NFS-exported area of the front end. An empty disk image is first created as a large sparse file, and then formatted for a Linux file system.
4. Optionally, empty swap images are created and stored in an NFS-exported area on the front end. An empty disk image is first created as a large sparse file, and then formatted as a Linux swap.
5. A configuration file for the hypervisor is created, and put into the NFS-exported area of the front end.
6. The front end sends a launch command through an SSH connection to invoke the new VM on the target node.
7. The new VM is launched using the configuration file and the disk and swap images on the NFS of the front

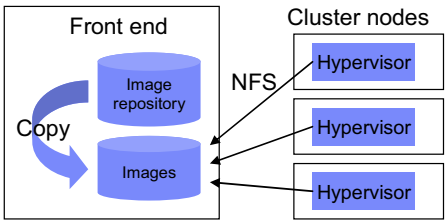


Figure 1: Architecture of OpenNebula NFS configuration

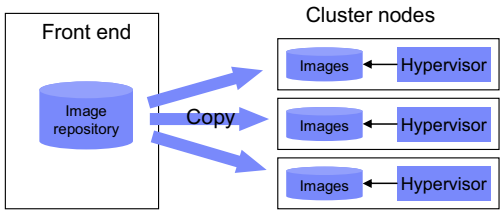


Figure 2: Architecture of OpenNebula LocalDisk configuration

end.

The benefit of this NFS configuration is quick provisioning when a small number of VMs are provisioned. Copying inside provisioning is faster. However, the disk access from the VM is slower. Each time the VM accesses its local disk, there is NFS access through the network because the local disks are actually located in the NFS exported area of the central front end.

Another benefit of the NFS configuration is live migration support. If disk images of each running VM are located on NFS, we can migrate a running VM to another cluster node without suspending its execution. This feature is very useful for the management and load-balancing of cluster nodes.

Raw disk images are pre-allocated as dense files, so the preparation of a disk image takes a long time. Operating system disk images tend to contain many zero-filed areas, so lazily allocating disk images as sparse files reduces the provisioning time. However, lazy allocation may cause performance degradation of the VM at runtime, because disk allocation must occur the first time the VM uses the unallocated disk.

2) OpenNebula LocalDisk configuration

The steps to provision a new VM in the OpenNebula LocalDisk configuration differ from those of the NFS configuration in these ways:

- In step 2, a root disk image is copied from the front end's image repository to the disk space of the cluster node via the network. The network copying is done with SSH's scp command.
- In Steps 3 and 4, an optional empty ephemeral disk and a swap disk can be created at the target cluster node.

Compared to the NFS configuration, the LocalDisk configuration does not have an obvious bottleneck at the central storage for the disk images of cluster nodes. The central image repository on the front end is read-mostly, and the disks of each cluster node are write-mostly during provisioning. This behavior improves the hit ratio of the file cache on the front end and makes the image transfers faster.

In the LocalDisk configuration, we cannot use live migration because the disk images are located on each cluster node, and it is hard to move them to another node quickly enough to transfer the execution of a running VM.

3) OpenNebula's support for eager and lazy disk allocations

As noted, OpenNebula uses standard (dense) files for the disk images that contain operating systems, and uses sparse files for the disk images of ephemeral data and swap data. This setting is customizable, so we prepared Eager and Lazy configurations for both the NFS and LocalDisk configurations. We prepared a total of four configurations:

- *NFS Eager*: all disk images are stored as dense files on the front end.
- *NFS Lazy*: all disk images are stored as sparse files on the front end.
- *LocalDisk Eager*: disk images are stored as dense files in the local disk of each VM.
- *LocalDisk Lazy*: disk images are stored as sparse files in the local disk of each VM.

To enable lazy disk allocation, we prepared the machine image as a sparse file. In the NFS configurations, the image is copied using the Linux cp command for provisioning, and the Linux cp command creates the copied file as a sparse file if the original file is sparse. This dramatically reduces the time for creating disk images, improving provisioning performance.

For network transfers of images in the LocalDisk configurations, the original OpenNebula uses the SSH scp command, which does not give sparse files special treatment. We customized OpenNebula to use the rsync command, which can more efficiently copy sparse files.

B. Eucalyptus

Eucalyptus provides cloud services compatible with Amazon EC2 and S3. It uses the Amazon API and the Amazon Machine Image (AMI) format. It additionally provides its own command line tools to manage the VMs.

We can run our own machine images of Amazon EC2 on Eucalyptus without modification. We can also use our own custom tools that use Amazon APIs by changing the base URL of the APIs. These are major advantages for Eucalyptus.

There are three main components in Eucalyptus: cloud controller, cluster controller, and node controller. The cloud controller provides the interface of the REST/SOAP-based APIs through which end users control their machine images and VMs. The cloud controller also provides storage services compatible with Amazon S3. The cloud controller controls the cluster controller based on end users' requests. The cluster controller manages the host machines that host VMs using the Xen hypervisor. In each host machine, the node controller runs and interacts with the hypervisor based on the commands from the cluster controller.

In contrast to OpenNebula, Eucalyptus only supports the LocalDisk configuration. User machine images are stored in the central storage service. When a user creates a new VM from an image, the image is restored in the local disk of a host machine, and the created VM uses the image as its local disk.

For placement policies, Eucalyptus supports *greedy* (the first node found will be used) or *round-robin* (a node will be selected in a round-robin fashion), but it does not support user-defined custom policies. Eucalyptus's round-robin policy is not identical to OpenNebula's striping policy because OpenNebula balances the number of running VMs among the nodes, but it only counts running VMs and ignores VMs during provisioning. Therefore, when we create multiple VMs at the same time, OpenNebula may have more imbalance than Eucalyptus.

Here are the steps to provision a new VM in Eucalyptus:

1. The cluster controller selects a target node based on the user's placement policy
2. The cluster controller checks the target node to see whether or not the machine images was previously provisioned and cached in the node. If it is cached in the target node, skip to Step 6.
3. The cluster controller extracts a machine image that contains a file system backup archived with the UNIX tar command and some metadata encrypted with the user's secret key. This includes operating system files and possibly user data and applications.
4. The cluster controller creates a new disk image from the extracted files and caches it on the cluster controller's local disk. In this step, it first creates a large empty dense file, formats the file for a Linux file system, and then copies the extracted files into the image.
5. The cluster controller transfers the created disk image from the cache to the target node.
6. Optionally, empty ephemeral disk images are created on the target node. An empty disk image is first created as a large dense file, and then formatted for a Linux file system.
7. Optionally, empty swap images are created and stored in the NFS-exported area of the front end. An empty disk image is first created as a large dense file, and then formatted as a Linux swap.
8. A configuration file for the hypervisor is created on the target node.
9. The cluster controller sends a launch request through the node controller to invoke the new VM on the target node.
10. The new VM is launched using the configuration file and the disk and swap images on the target node.

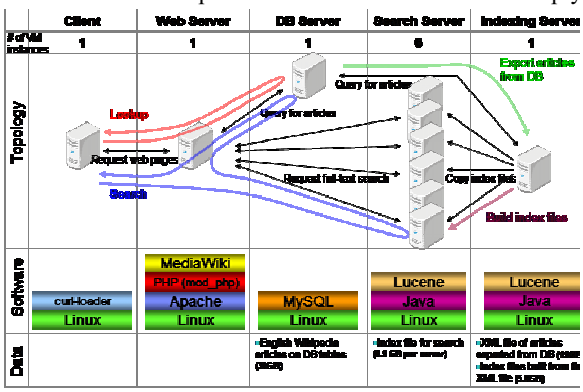


Figure 3: Architecture of the Wikipedia workload

The caching mechanism is an advantage of Eucalyptus. It is quite helpful in creating multiple VMs from a single machine image.

One of the major limitations of Eucalyptus is the lack of lazy disk allocation. It always uses dense files, so preparing disk images is slow and provisioning tends to take long time. Another disadvantage is its limited customizability. Most of the features are hard-coded in the source code, and the behaviors cannot be modified easily.

C. *Amazon EC2*

In Amazon EC2, we can manage VMs using the Amazon EC2 API. Amazon also provides Web-based management console, and command-line tools. Machine images are stored on Amazon Simple Storage Service (S3) as the Amazon Machine Image (AMI) format. Amazon EC2 employs the Xen hypervisor for virtualization.

Amazon EC2 does not disclose its internal architecture, so we cannot describe its configuration and provisioning sequence.

IV. WORKLOADS

To evaluate the open-source cloud platforms, we prepared two realistic and representative workloads. In this section, we describe our workloads in detail.

A. *Wikipedia*

The Wikipedia websites use the LAMP (Linux, Apache, MySQL, and PHP) architecture, along with Java for search engine support. This is a typical Web application and complex enough to assess the cloud platforms. We used all of the articles from the English version of Wikipedia, which is enough data for a meaningful evaluation.

We used 9 VMs to provide the Wikipedia services, and 1 VM as the client. The 9 VMs include a Web server, a DB server, an index server, and 6 search servers. A total of 10 VMs running Linux were provisioned for our evaluations.

We installed the Apache HTTP Server and an Apache module for PHP (mod_php) on the Web server.

MediaWiki is a Wiki server written in PHP. This Wiki server is the main component of the Wikipedia system. There is no actual data in this server, but the article data is fetched from a backend DB server. Memcached also runs on this server to cache article data from the DB. For search engine support, we installed a MediaWiki plugin for the Lucene search engine. This plugin transfers search requests to the

backend search servers.

In the DB server, MySQL manages the Wikipedia article data. The MySQL table that contains all of the English Wikipedia articles is about 30 GB.

The search servers use the Lucene search engine. Lucene is written in Java, and a Java VM is needed for the search engine software. The index file for the search engine is divided among 6 servers, and the searches are balanced among these servers. The index file is generated by the backend index server.

The index server runs the indexer for Lucene. The article data is exported from the DB server and stored as a single XML file of about 19 GB. To update the indexes, the indexer parses the XML file and generates 6 index files, and then copies them to the search servers.

In the client, we used the HTTP load generator curl-loader, which generates queries for articles as HTTP requests and submits the search requests.

To evaluate the performance of the cloud platforms, we measured four performance metrics. Two of them are the throughputs of Web transactions, and the other two are the elapsed times for batch processing tasks.

The first throughput metric is *Lookup*. This is the throughput for HTTP requests to fetch Wikipedia articles generated from the client. The lookup key words are 100,000 words that were randomly selected from the article data.

The second throughput metric is *Search*. This is a throughput for HTTP requests to search Wikipedia articles using the Lucene search engine. The search key words are 100,000 words that were randomly selected from the article data.

The first batch metric is *Export*. This is the elapsed time to export all of the article data from MySQL to an XML file.

The second batch metric is *Index*. This is the elapsed time to create the index files from the exported article data.

B. DayTrader

Apache DayTrader [19] is a benchmark application built on Java EE technologies such as Java Servlets, JavaServer Pages for the presentation layer, and JDBC, Java Message Service, Enterprise JavaBeans and Message-Driven Beans for the back-end business logic and the persistence layer. This benchmark simulates an online stock trading system with multiple client users.

We used this benchmark to evaluate the performance interference in a multi-tenancy scenario. In the Wikipedia scenario, there is a single user that uses a target cloud system. However, typical cloud systems accommodate multiple users, and VMs of different users coexist in a host machine. To evaluate this situation, we built a workload that runs multiple DayTrader systems in a target cloud.

Each DayTrader system consists of Web and DB servers. IBM WebSphere Application Server V7 is installed in the Web server and a second server has IBM DB2 V9.7. Both servers are prepared as custom images based on Red Hat Enterprise Linux 5.4.

In our scenario, each cloud user requests a pair consisting of of a Web server and a DB server, and up to 16 users request a DayTrader system. The requests are processed one by one, so each user's request is processed after previous user's request is finished. When a user's request is completed, the DayTrader benchmark begins to generate HTTP traffic to stress the DayTrader system of that user. This means the VM provisioning and the DayTrader benchmark run in parallel.

V. PERFORMANCE EVALUATION

We measured the cloud platforms using the same hardware and network equipment for fair comparisons. We also tested our workload on Amazon EC2, and compared the performance of EC2 to the open-source platforms by compensating for the CPU speeds of the VMs. To do this adjustment, we first measured the SPEC CPU benchmark on both our hardware and Amazon EC2.

A. Hardware environment

We used 5 identical physical servers. One is used for the central front end, and the other four are used as host machines.

Our servers are IBM BladeCenter HS22, which has dual sockets with Intel Xeon E5570 2.93 GHz processors. A single Xeon processor has 4 cores, and each core has 2 hardware threads. A total of 16 hardware threads are available in a single HS22. Each server had 24 GB of memory installed, and a 10-Gb Ethernet card. For storage, each server has a 32-GB SATA SDD and a 500-GB SATA HDD.

Red Hat Enterprise Linux 5.4 is installed on the SDD of each server. The cloud platform software is also installed in the SDD, but the images of the VMs are stored on the HDD.

These blade servers were in a single IBM BladeCenter H chassis and connected via Nortel's 10-Gb Ethernet Switch.

B. SPEC CPU2006

Amazon EC2 uses EC2 compute units for pricing. The prices for Amazon EC2 VMs are proportional to the processing power of the VMs as measured in EC2 compute units.

To compare the performance between Amazon EC2 and our hardware environment, we ran SPECcpu2006_int on both EC2 and our environment. We rented a small EC2 instance, with just 1 EC2 compute unit. Our measurement indicated that our hardware has 3.39 EC2 compute units in each hardware thread.

Then we adjusted the CPU processing power allocated to

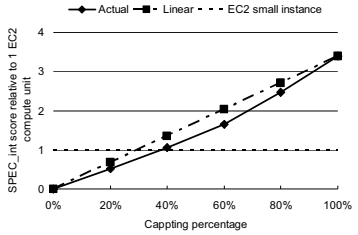


Figure 4: SPEC CPU score with Xen's capping

each VM using the capping feature of the Xen hypervisor. First we used 1/3.39=30% capping, but the processing power was smaller than expected. The actual relationship between capped processing power and capping percentage is not linear. Figure 4 shows the actual relationship we found. This result indicates that we need to use 40% as the capping percentage to obtain 1 EC2 compute unit with our hardware.

C. Scalability of provisioning

As described in Section III, the configuration of the cloud platform affects the performance. We measured the performance in provisioning VMs on the cloud platforms with OpenNebula's four configurations, Eucalyptus, and Amazon EC2. We provisioned 1, 2, 4, 8 and 16 VMs from a single machine image. Each VM has a 10-GB main disk, a 10-GB ephemeral disk, and a 4-GB swap disk.

Figure 5 shows the elapsed time for provisioning the VMs. In the 1-VM case, Eucalyptus was slowest because Eucalyptus creates dense files for the ephemeral and swap disks, and also caches the OS images. Caching is effective when we create more than one VM from a single image, but in this case it is only overhead. OpenNebula NFS Lazy was fastest since only sparse files were copied inside the front end.

From 2 VMs to 4 VMs, OpenNebula NFS Eager was becoming worse, but OpenNebula LocalDisk Eager and Eucalyptus were unchanged. In the NFS configuration, both the image repository and storage for the VMs are located in the central server and are shared among the VMs, so this becomes a bottleneck. In the LocalDisk configuration, the image repository is also located in the central server and shared, but disk images of the VMs are located in the host machines. This means the disk writing for provisioning is distributed among host machines and avoids I/O contention.

Above 4 VMs, OpenNebula NFS Eager was worst. OpenNebula LocalDisk Eager and Eucalyptus also declined. This is because we had 4 host machines, so disk writes cause contention even if the VMs are balanced among the host machines. We see improvements due to the lazy disk allocation on the OpenNebula NFS Lazy configuration. This dramatically reduced the times for creating disk images and improved the provisioning performance.

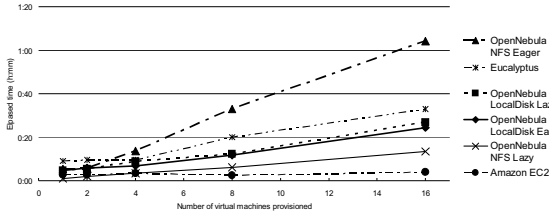


Figure 5: Provisioning times

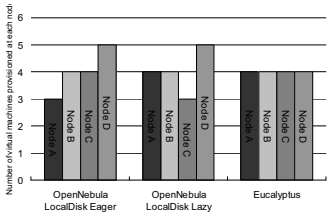


Figure 6: Number of VMs provisioned at each node

As we mentioned earlier, OpenNebula does not support provisioning multiple VMs at the same time, so we need to provision them one at a time. Each time a user submits a create request, it is queued, and the system checks the queue every 30 seconds. If there are any requests in the queue, the system calculates the priority for each node based on the user's definition and chooses the node with the highest priority. Then the system starts provisioning at the selected node. If there are multiple create requests in the queue, all of them will be provisioned at the same selected node. This creates imbalance when we submit multiple create requests within the 30-second interval. To avoid this problem, we should not submit more than one request within 30 seconds. When we need to provision 16 VMs, we need to wait 15 × 30 seconds = 450 seconds to submit the request for the last VM.

The placement policy of OpenNebula is not pure round-robin. The priority of each node is calculated from the number of running VMs on it. The number does not include VMs that are being provisioned. Since provisioning a VM does not usually finish within the 30-second interval, the number of VMs that have been requested may not be reflected in the priority calculation.

Figure 6 shows the numbers of VMs provisioned at each node when we conducted Wikipedia experiments on 16 VMs using OpenNebula LocalDisk Eager, OpenNebula LocalDisk Lazy, and Eucalyptus. We used 4 physical machines for the cluster nodes, so each bar in the figure represents the number of VMs provisioned at one of the 4 nodes. With Eucalyptus, there were exactly 4 VMs provisioned on each node. On the OpenNebula LocalDisk Eager and Lazy configurations, the provisioning of the VMs was not balanced.

D. Wikipedia workload

We ran the Wikipedia workload on OpenNebula and Eucalyptus. For OpenNebula, we used the LocalDisk Eager, LocalDisk Lazy, and NFS Eager configurations. The test of OpenNebula NFS Lazy was not done because the results of LocalDisk Lazy and NFS Eager indicated that the test for NFS Lazy would take too long to complete.

Table 1 shows the resource sizes defined for each VM in our Wikipedia workload. The operating system of these VMs is CentOS 5.3 (64-bit). We prepared root disk images for CentOS, and installed the required software described in Section A in the images, storing them in the image repositories of the central server. The time to install the required software and to configure the software is not included in the performance results. Each of the root disk images is 10 GB, and a 4-GB swap disk was used for each VM. The rest of the disk space on each VM is used for data as an ephemeral disk. On the DB server, the ephemeral disk is used for storing the DB tables that contain the Wikipedia articles. On the search

servers, the ephemeral disk is used for storing the index file for the search engine. On the index server, the ephemeral disk is used to store the exported article file, and the generated index files. For the other servers, the ephemeral disks were not used.

1) Provisioning

Figure 7 show the provisioning times for VMs created in the Wikipedia workload. Lazy disk allocation offers faster provisioning than Eager allocation of OpenNebula and Eucalyptus. The OpenNebula LocalDisk Lazy configuration was 2.7 times faster in the provisioning than the OpenNebula LocalDisk Eager on a geometric-mean average.

Eucalyptus was 21% slower than OpenNebula LocalDisk Eager in the provisioning on a geometric-mean average. This is mainly because Eucalyptus's caching mechanism was not effective on our workload and became just overhead. We did an additional experiment that provisioned a Client server at the host machine where another Client server was already provisioned. This experiment shows 16% faster provisioning when caching is used.

OpenNebula NFS Eager was 75% faster in the provisioning than OpenNebula LocalDisk Lazy by the geometric mean. This is because copying disk images inside a central server is faster than transferring them via network. However the performance improvement is smaller than with lazy-disk allocation.

2) Batch processing

Our experiments showed that lazy disk allocation for tasks that did many disk writes caused serious performance degradation in the first execution because of the overhead for disk allocation for first writes. Figure 8 shows the elapsed times of Export and Import. In OpenNebula LocalDisk Eager was 43% faster than that with OpenNebula LocalDisk Lazy on a geometric mean average of Export and Import.

To measure the effect of lazy disk allocation, we also ran Export and Index on the OpenNebula LocalDisk Lazy configurations after the first run completed. Figure 8 also includes the results of the second runs. The second runs were faster than the first runs. This is because the disk writes of the first run caused disk allocation, and reduced the necessary disk allocation of the disk writes in the second runs. Actually, the elapsed time of the second run of Export with lazy allocation was almost equal to that with eager allocation. The second run of Export overwrote the exported file generated by the first run, so there was no need to allocate new disk areas. In contrast, the elapsed time for the second run of Index with lazy allocation was shorter than for the first run, but still longer than with eager allocation. The index files generated by the first run were deleted, and the second run created new index files. The disk areas for the new index files generated by the second run were not necessarily in the same disk areas as the index file generated by the first run, so some of the disk areas for the second run were new areas, and they caused additional disk allocations. These additional disk allocations caused the performance degradation compared to eager allocation.

Eucalyptus was only 0.4% faster for Export than OpenNebula LocalDisk Eager. This result was expected because both use local disks with eager-allocation. For Index, however, Eucalyptus was 15% faster than OpenNebula LocalDisk Eager. We are still investigating the causes of this improvement.

OpenNebula NFS Eager showed very poor performance for the batch processing. The geometric mean was 2.9 times slower than OpenNebula LocalDisk Eager on a for Export and Index. In the Export task, Lucene exports the article data from the DB server, and stores it on the Index server, so reading from the DB server's disk, and writing to the Index server's disk caused contention in the NFS-exported areas of the front end server. In the Index task, Lucene reads the exported article data and generates the index files. Both reading and writing these files were done inside the Index server, but I/O operations went thought the network, and caused performance

Table 1: Resources of virtual machines of the Wikipedia workload on OpenNebula and Eucalyptus

	Client	Web	DB	Search	Index
VMs	1	1	1	6	1
Processor	2	4	2	1	2
Memory	4 GB	8 GB	4 GB	2 GB	4 GB
Disk	48 GB	72 GB	72 GB	24 GB	48 GB

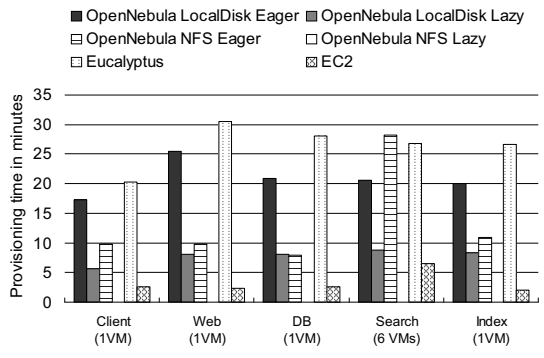


Figure 7: Provisioning times of the virtual machines for the Wikipedia workload

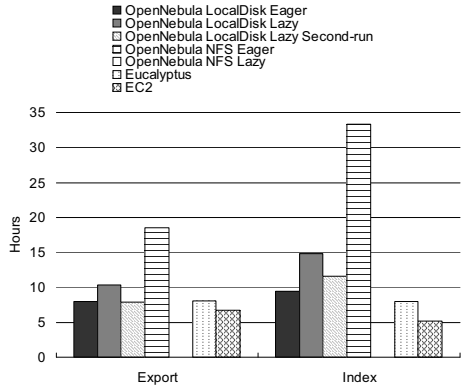


Figure 8: Elapsed time of Export and Index

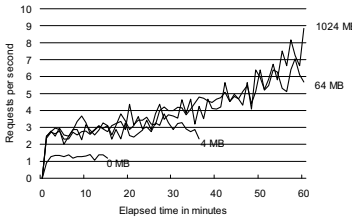


Figure 9: Throughputs of OpenNebula LocalDisk Lazy with memcahced with 0, 4, 64, and 1024 MB buffer size.

degradation. If these are daily batch jobs, this would beg unacceptable because the execution times exceed 24 hours.

3) Web transactions

After we created the index files, we measured the throughputs for Web transactions. We measured the throughputs for 10 minutes after we warmed up for 20 minutes.

For the measurement of the throughputs, we set the size of the memory buffer for memory cached at the Web server to 4 MB. This was intentionally small to reduce the measurement times. Figure 9 shows the throughputs of Lookup on the OpenNebula LocalDisk Lazy file configuration with cached memory buffer sizes of 0, 4, 64, and 1024 MB. The 0 MB means that memory caching was not done. For the 0-MB case, the throughput was saturated after 5 minutes. For the 4-MB case, the throughputs were saturated after 20 minutes. However, the throughputs were not saturated for the 64- and 1024- MB cases within one hour. This result indicated that we need to run more than one hour to get steady states when measure for 64- and 1024-MB cases. To minimize the measurement time, we chose the 4-MB configuration for our measurements. Obviously, this is not large enough to accommodate data of all Wikipedia articles that were retrieved during measurement, so there were always memory cache misses, resulting in disk reads. This means that I/O performance significantly affected the result performance scores.

Figure 10 shows the throughputs of Lookup and Search on OpenNebula LocalDisk Eager, OpenNebula LocalDisk Lazy, OpenNebula NFS Eager, and Eucalyptus. The absolute throughput values are not so high because we used very small buffers for memcached. The throughputs are similar values between OpenNebula LocalDisk Eager and Lazy configurations. This is because disk I/O of the transactions are almost all read-only, so the penalty of disk writes for lazy

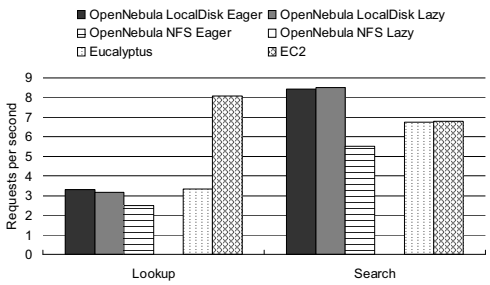


Figure 10: Throughputs of Lookup and Search

allocation did not affect the throughputs on OpenNebula LocalDisk Lazy. We did not see any difference for Lookup between OpenNebula LocalDisk Eager and Eucalyptus either, because both use eager allocation.

For Search, Eucalyptus had lower throughput than the others. We are investiating the causes.

OpenNebula NFS Eager was 50% slower than OpenNebula LocalDisk Eager by the geometric means for Lookup and Search. The performance degradation was larger in Search than Lookup. Lookup only reads the DB tables on the DB server, but Search reads both the DB tables on DB server and the index files on Search servers, so Search caused I/O contention at the NFS-exported areas of the front-end server.

In most cases, the throughput for Web transactions is the most important metric for websites, but the NFS configuration showed the worst throughputs in our measurements, so the NFS configuration should be avoided for this kind of workload. The NFS configuration is only useful when the provisioning performance is more important than Web transactions and batch processing tasks.

E. Comparison with Amazon EC2

We also ran the same Wikipedia workload on Amazon EC2. The hardware resources used on EC2 are different from the open-source experiments, because EC2 supports large VMs, while our test environment provides only small VMs. As described in Section B, we measured the SPEC CPU on both EC2 and our hardware, and capped the processing power of our hardware with Xen's capping to make our virtual processor match 1 EC2 compute unit. Table 2 shows the hardware resources for the VMs we used in EC2.

Figure 7 includes the provisioning times on EC2, and show that EC2 had the fastest provisioning for every server.

Figure 8 includes the elapsed time for Export and Index on EC2. Again, EC2 offered better performance for both Export and Index. These tasks are disk-I/O intensive, and we used slow SATA disks, so this may have reduced the performance in our environment.

Figure 10 includes the throughputs for Lookup and Search on EC2. For Lookup, EC2 had more than twice the throughput compared to OpenNebula and Eucalyptus. In addition to our slow disks, this is because the DB on EC2 server has more memory, and the DB tables were cahed more efficiently than in our environment. In contrast, for Search, the throughput on EC2 is similar to the others. Search is less disk-I/O intensive than Lookup because it fetches smaller amounts of data from the DB, and part of the index file is stored in the Java heap of the Java VM on each search server.

Table 2: Virtual machines resources for the Wikipedia workload on Amazon EC2

	Client	Web	DB	Search	Index
VMs	1	1	1	6	1
Processor	2	4	2	1	2
Memory	7.5 GB	15 GB	7.5 GB	1.7 GB	7.5 GB
Disk	850 GB	1,690 GB	850 GB	160 GB	850 GB

We conducted additional experiments with faster HDDs on our test environment. We replaced the 7,200-RPM SATA HDDs with faster 15,000-RPM SAS HDDs on our test machines, and measured the throughputs of Lookup and Search. We obtained 8.5 requests per second for Lookup, and 16.0 requests per second for Search. In both cases, our test environment with the faster HDDs outperformed the EC2 results, so our experiments indicate that the performance of the HDDs greatly affects the performance.

F. DayTrader workload

We ran the DayTrader scenario on Eucalyptus using the 4 host machines. We provisioned up to 16 DayTrader systems, each of which consists of a pair of a Web server and a DB server, so a total of 32 VMs were provisioned in this experiment. The provisioning of pairs of VMs is done one pair at a time. That is, after provisioning a pair of VMs , provisioning of another pair can be started. Just after provisioning of a VM pair is completed, generation of web traffic to the provisioned DayTrader system starts. Therefore, this Web traffic might interfere with the performance of subsequent provisioning.

Figure 11 shows the times to provision DayTrader systems. Each dot shows a time to provision a pair of servers for a user. The solid line shows the provisioning time for the benchmark's Web traffic. The dotted line shows the provisioning time without the benchmark's web traffic.

In the results with traffic, the interference from the Web traffic may increase as the number of users increases, but we saw no large differences between these two cases. This means that activities of the users' VMs did not significantly affect the provisioning time.

In contrast, throughputs of the Web servers were strongly affected by the provisioning. Figure 12 shows the throughputs per second of the first and second users. In this graph, only two users are shown. There were a total of 16 users, but the other 14 users are not shown to keep the graph simple. There are spikes of throughputs due to provisioning activities. The timing of the spikes for the first and second users are offset from each other. This is because there are 4 host machines, so provisioning the Web and DB servers only affected two of the 4 host machines at the same time.

Figure 13 shows the CPU and disk usage of host machines 1 and 2. Host machine 1 hosted up to 8 web servers, and host machine 2 hosted up to 8 DB servers. The CPU utilization graphs indicate that VMs were added one by one, and the CPU utilization dropped during VM provisioning. We capped 1 core per VM in this experiment, so each VM only consumed up to 100% of 1 CPU. This left CPU resources available for newly created VMs, so the performance degradation during provisioning was due to disk contention, as shown in the graph of DiskBusy Percentage.

CPU capping works well to conserve CPU resource for newly provisioned VMs, but there is no mechanism to preserve disk I/O capacity for specific VMs. To offer

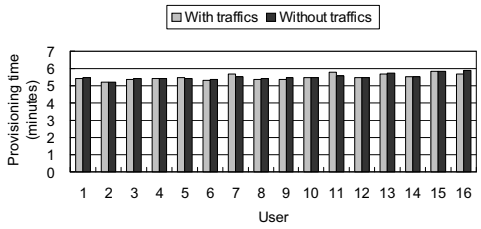


Figure 11: Time to provision DayTrader systems

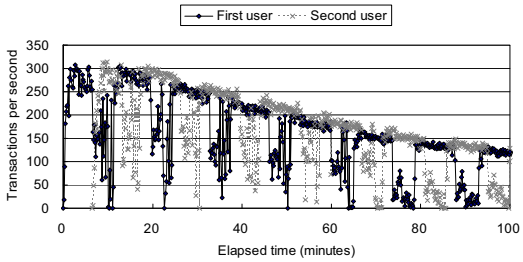


Figure 12: Throughputs for the first and second users

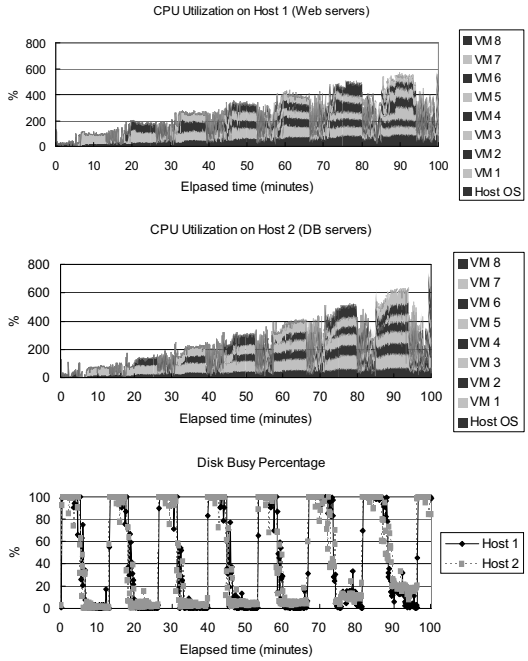


Figure 13: CPU and disk usage of host machines 1 and 2

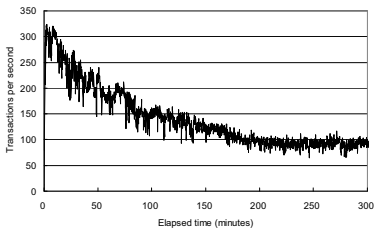


Figure 14: Transactions per second of single-user case

quality-of-service or service-level agreement for cloud users, capping the disk I/O capacity for provisioning is necessary to avoid interference with the performance of running VMs.

The throughputs of each user gradually decreased as the number of users increased. This is a known characteristic of the DayTrader benchmark. Figure 14 shows the throughput of a single-user case for a long run. There was no interference from other VMs. Even in this case, the throughput gradually decreased, and it took more than 3 hours to reach the stable state.

VI. FUTURE WORK

In our research, we tried some experimental over-commitments for the processors and disks. Processor over-commitment is CPU capping. We used 40% capping for the CPU, so we could provides 2.5 times more virtual CPUs than physical CPUs. Lazy disk allocation is disk over-commitment. We can assign more disk space for VMs than physical disk space when some of VMs do not use all of their disk space. However, we did not do any experiments with memory over-commitment because none of our target platforms directly supports memory over-commitment (though Xen does support it). We are currently working on experiments using Xen's memory ballooning feature. We are also planning to evaluate other memory over-commitment features available in VMware such as content-based memory sharing [16].

VII. CONCLUSIONS

We evaluated the performance variations of workloads using the open-source cloud platforms OpenNebula and Eucalyptus with a Wikipedia workload, and found two important configuration choices for the cloud platforms: (1) The physical location of the VM disk images (local disk or NFS), and (2) eager or lazy allocation of the VM disk images.

We measured (1) the provisioning times of the VMs, (2) the elapsed times of two types of batch processing, and (3) the throughputs of two types of Web transactions with the Wikipedia workload. The local-disk configuration performed 75% slower in the provisioning, 2.9 times faster for batch possessing, and 50% faster for the Web transactions than the NFS configuration. The eager-allocation configuration was 2.7 times slower in the provisioning, 43% faster for the batch processing, but only 1.5% faster for the Web transactions than the lazy-allocation configuration. Our results indicate that no configuration performs the best performance for all of the three metrics at the same time, so the best choice is vary depending on which metric the cloud user focuses on.

If batch processing is more important than provisioning, the local-disk configuration with eager disk allocation should be used. Otherwise, local-disk allocation with lazy allocation should be used. The NFS configuration is suitable only when provisioning time is more important than the other metrics.

We also evaluated a multi-tenancy scenario using the Apache DayTrader Benchmark, and the results showed that provisioning significantly affected the throughputs of DayTrader due to the lack of a disk-I/O capping mechanism.

REFERENCES

[1] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM, New York, NY, 164-177.

[2] Heo, J., Zhu, X., Padala, P., and Wang, Z. 2009. Memory overbooking and dynamic control of Xen virtual machines in consolidated environments. In Proceedings of the 11th IFIP/IEEE international Conference on Symposium on integrated Network Management (New York, NY, USA, June 01 - 05, 2009). Institute of Electrical and Electronics Engineers The, 630-637.

[3] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[4] Binnig, C., Kossmann, D., Kraska, T., and Loesing, S. 2009. How is the weather tomorrow?: towards a benchmark for the cloud. In Proceedings of the Second international Workshop on Testing Database Systems (Providence, Rhode Island, June 29 - 29, 2009). DBTest '09. ACM, New York, NY, 1-6.

[5] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. 2009. The Eucalyptus Open-Source Cloud-Computing System. In Proceedings of the 2009 9th IEEE/ACM international Symposium on Cluster Computing and the Grid (May 18 - 21, 2009). CCGRID. IEEE Computer Society, Washington, DC, 124-131.

[6] D. Schanzenbach and H. Casanova, "Accuracy and Responsiveness of CPU Sharing Using Xen's Cap Values," Computer and Information Sciences Dept., University of Hawaii at Manoa, Tech. Rep. ICS2008-05-01, May 2008.

[7] Sotomayor, B., Montero, R. S., Llorente, I. M., and Foster, I. 2009. Virtual Infrastructure Management in Private and Hybrid Clouds. IEEE Internet Computing 13, 5 (Sep. 2009), 14-22.

[8] Amazon EC2. http://aws.amazon.com/ec2/

[9] Nimbus Project. http://www.nimbusproject.org/

[10] Apache VCL. http://cwiki.apache.org/VCL/

[11] Averitt, S., Bugaev, M., Peeler, A, Shaffer, H, Sills, E, Stein, S., Thompson, J., Vouk, M. Proc. International Conference on Virtual Computing Initiative, May 7-8, 2007, IBM Corp., Research Triangle Park, NC, pp. 1-16.

[12] MediaWiki. http://www.mediawiki.org/wiki/MediaWiki

[13] Apache Lucene. http://lucene.apache.org/

[14] Wikipedia. http://www.wikipedia.org/

[15] Kernel-based Virtual Machine. http://www.linux-kvm.org/

[16] Waldspurger, C. A. 2002. Memory resource management in VMware ESX server. SIGOPS Oper. Syst. Rev. 36, SI (Dec. 2002), 181-194.

[17] Curl-loader. http://curl-loader.sourceforge.net/

[18] Vahalia, U. 1996. UNIX Internals: the New Frontiers. Prentice Hall Press

[19] Apache Geronimo DayTrader Benchmark. https://cwiki.apache.org/GMOxDOC20/daytrader.html

Runtime Workload Behavior Prediction Using Statistical Metric Modeling with Application to Dynamic Power Management

Ruhi Sarikaya, Canturk Isci, and Alper Buyuktosunoglu
IBM Thomas J. Watson Research Center, NY
{sarikaya,canturk,alperb}@us.ibm.com

Abstract

Adaptive computing systems rely on accurate predictions of workload behavior to understand and respond to the dynamically-varying application characteristics. In this study, we propose a Statistical Metric Model (SMM) that is system- and metric–independent for predicting workload behavior. SMM is a probability distribution over workload patterns and it attempts to model how frequently a specific behavior occurs. Maximum Likelihood Estimation (MLE) criterion is used to estimate the parameters of the SMM. The model parameters are further refined with a smoothing method to improve prediction robustness. The SMM learns the application patterns during runtime as applications run, and at the same time predicts the upcoming program phases based on what it has learned so far. An extensive and rigorous series of prediction experiments demonstrates the superior performance of the SMM predictor over existing predictors on a wide range of benchmarks. For some of the benchmarks, SMM improves prediction accuracy by 10X and 3X, compared to the existing last-value and table-based prediction approaches respectively. SMM's improved prediction accuracy results in superior power-performance trade-offs when it is applied to dynamic power management.

1 Introduction

Today's microprocessors rely on adaptive power and performance management schemes that try to adapt to changing behavior of applications. Typically, these schemes exploit changing phase behavior at certain time granularities to dynamically trade-off power and performance. In most cases, the techniques are reactive in that they are enabled once the phase transition occurs. In cases where the application phase behavior is very dynamic, reactive systems can result in poor performance and may lead to possible oscillations in the employed adaptive control.

Ideally, one would like to predict future behavior of an application based on its past behavior and proactively manage the system to avoid instability. Obviously, such a scheme heavily relies on accurate prediction of application behavior. Prior work proposed such proactive methods based on pattern history tables [16], program flow behavior [22, 20], and statistics over recent performance characteristics [10]. These techniques predict application behavior by tracking recently observed patterns or statistics in the observed application features, which can be used to guide dynamic management decisions.

While in general these predictors prove to be effective in many scenarios, there are two critical caveats due to i) their deficiency in predicting global long range patterns, and ii) their inability to model patterns of varying length. Figure 1 shows an example to this with an actual execution trace from the `applu` benchmark. Here, the two boxed regions show an exemplary repetitive execution. The table based predictor used in this example has a pattern length of 8 samples. The two captured repetitive regions show an 8-sample pattern, where the second occurrence experiences a single fluctuation. While the table based predictor learns the behavior following from the first pattern, this small fluctuation actually leads to a pattern mismatch. As a result, the table based predictor backs off to last-value prediction, and the following two consecutive samples labeled as *i* and *j* are mispredicted. However, an advanced predictor that is resilient to pattern fluctuations, can actually discern this repetition and can correctly predict both *i* and *j* from prior observations. This pattern discovered by our proposed predictor—by modeling varying size patterns—is highlighted with the smaller enclosed regions in the figure. Our proposed predictor can be effective over both table based and other historical predictors by modeling patterns of varying length and by tracking global application patterns.

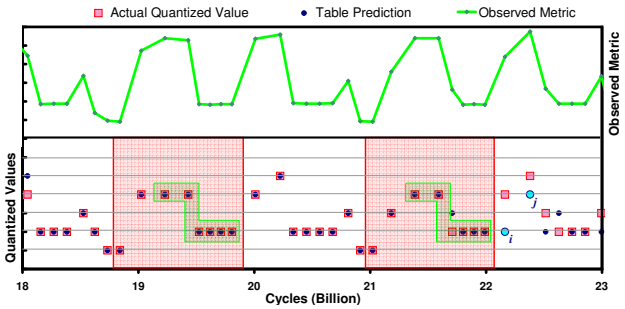


Figure 1. Vulnerability of table based predictor to fluctuations in observed metrics.

In this paper, we propose a Statistical Metric Model (SMM) for metric prediction. The SMM estimates the probability of a finite a sequence. The probabilities generated by the SMM are used to predict the most likely next phase based on prior observations. SMM alleviates the shortcomings of the prior predictors with its ability to model patterns of different length and long-term global patterns. A comprehensive set of experiments demonstrates the effectiveness of the SMM predictor in comparison to the previously proposed predictors. SMM predictor, when applied to dynamic power management, results in better power-performance trade-offs compared to the existing predictors.

The rest of the paper is organized as follows. Section 2 gives an overview of prior work on program phase prediction.

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

Section 3 describes the baseline predictors that SMM is compared against. Section 4 provides a description of the foundation and the formulation of the SMM predictor. Section 5 summarizes our methodology. Section 6 demonstrates the experimental evaluation of SMM and its application to dynamic power management, and Section 7 offers our conclusions.

2 Related Work

A large body of research focuses on tracking, characterizing and predicting application characteristics. These studies leverage various characterization metrics including performance monitoring counters, programmatical flow and system statistics. One line of research explores characterization of observed behavior via runtime statistic collection and architectural or system-level simulations [14, 15, 21, 1, 3, 9, 12]. These techniques mainly focus on interpreting specific workload execution behavior and detecting some indicative characteristics. Other work [2, 4, 8, 5, 24] uses system statistics to guide dynamic adaptations such as power and thermal management. While these techniques provide significant insights to workload behavior and its impact on dynamic management decisions, they do not explore online prediction of future behavior. The resulting dynamic management techniques can be considered as reactive approaches. Our proposed approach aims to provide the necessary means for proactive adaptations.

In addition to the characterization studies, significant number of studies also target at predicting future application behavior [10, 16, 13, 22, 18, 19, 26]. Duesterwald et al. [10] describe different statistical and table based predictors for within- and across-metric predictions of performance monitoring information. They show that the table-based predictor generally outperforms the other predictors they tested. Sarikaya et al. [19] describe an optimal prediction technique based on a predictive least squares minimization. Isci et al. [13] develop a table based runtime predictor to predict future behavior from past pattern characteristics. Zhou et al. monitor memory access patterns and estimate memory behavior of workloads for energy efficient memory allocation [26]. Sherwood et al. describe microarchitectural phase predictors based on repetitive program flow behavior [22]. Shen et al. detect such repetitions from reuse distance patterns for dynamic memory configurations via profiling and instrumentation [20]. In summary, all these studies provide useful prediction techniques suitable for different applications. This paper, on the other hand, shows the benefit of statistical metric modeling for tracking varying pattern history lengths and modeling long term patterns.

3 Baseline Predictors

This section gives a brief overview of the predictors that serve as the baseline in our work.

3.1 Last Value Predictor

One simple, yet effective metric predictor is the last value predictor. Last value predictor assumes that the next metric sample is the same as the last observed sample. The last value predictor can be very accurate for slowly varying metric sequences. However, its performance suffers greatly for rapidly varying metrics. The main appealing feature of the last value predictor is its simplicity from the computational and storage perspective. Other more elaborate predictors introduce computational and storage overheads to the prediction process.

3.2 Table Based Predictor

Table based predictors track the patterns in prior application history to deduce future workload characteristics. Such approaches rely on the repetitive application execution characteristics to produce a reliable model of future behavior. An example of these predictors, namely a global phase history table predictor [13] is depicted in Figure 2. Such a table based predictor can be implemented either in software or hardware depending on the desired prediction quantum and policy application time granularities [13, 10, 22].

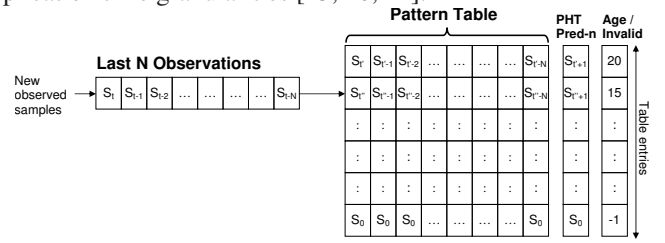


Figure 2. Table based predictor structure.

The table based predictor consists of a global register that tracks a certain number, N, of most recently observed sample characteristics. At each sampling period, this register records the last observed sample. The contents of this register are used to index into a pattern table, which holds a certain number of previously encountered patterns. The predictions are also deterministically encoded into this table and performed for each previously observed pattern. An age entry is used to track the reuse time of different entries for a least recently used (LRU) replacement policy. After a prediction is performed, the register contents are added to the table by either replacing the oldest entry or by inserting into an available invalid entry.

4 Statistical Metric Model

4.1 Intuition for SMM

The SMM is inspired from the way natural language is generated. In a natural language we use words to construct sentences. The SMM treats the metric samples as the words in a language and builds a language model for each metric. Note that, as metric samples are real numbers one has to quantize the real numbers into a set of discrete values, which are called "quantization bins". In a natural language, words do not follow each other randomly because of the underlying grammar. There is an underlying structure defined by the grammar, which determines the order in which we bring words together to make meaningful sentences. Our intuition is that, like in natural languages, we can treat the metric modeling as a language modeling problem. We assume that there is an underlying structure in each metric, and if indeed there is such an underlying structure (e.g. repetitive patterns) SMM can reveal and model this structure. However, if there is not any structure, that is to say, the metric is a completely random sequence of numbers then SMM will not do a worse job than any other predictor, as long as it is trained on sufficiently large data. In the

rest of the manuscript we will often draw parallels between the natural language modeling and the SMM to explain the concepts used in this work.

4.2 Foundations of SMM

We consider a metric as a sequence of quantized sample values. From now on, we refer to a "quantized sample" simply as "sample". The SMM is a conditional distribution on the identity of the ith sample in a metric sequence, given the identities of all previous samples. In other words, the next sample in a metric depends on all the previous samples. In general, this is a true statement but such a model will suffer from parameter estimation problems. Therefore, we have to make a computationally convenient approximation that a sample depends only on the previous n samples, where n depends on the amount of available data to estimate the model parameters. Again, going back to natural language modeling analogy, in general, what word we will speak next, depends more on the most recent previous n words than the words we have spoken a while ago.

The SMM we use is based on a class of Markov models, which is known as, the n-gram models [17]. N-gram models have received intensive research since their invention and have been widely used in speech and natural language processing [7]. The parameters of n-gram models are estimated from a large training text. The models produce a reasonable nonzero probability for every word in the vocabulary. From a theoretical perspective SMM is identical to the n–gram models. Here n refers to the maximum length of the finite sequence of the metric samples (e.g. patterns of $n = 4$ samples as given in Figure 3). The probability of the nth sample is conditioned on the previous $n-1$ samples. Unlike natural language models, which are built offline only once and then used for the application without any update, the SMM is updated as many times as the observed metric samples.

The SMM model of order $n = 4$ is shown in Figure 3. The model has two set of entries: the finite sequences and the associated probabilities with each sequence. The model contains sequences of length 1 to n where the last sample in each sequence is the output given the remaining $n-1$ history samples. For example, the first entry has the (s_1, s_2, s_3) as the history for the next sample s_4 with the probability $P(s_4|s_3, s_2, s_1)$. The probabilities are called the model parameters that are estimated from the observed data. As we highlighted before, the SMM consists of models of lower order m $(1 \leq m \leq n)$, increasing the likelihood of finding a matching subsequence for a given finite sequence.

There is an inverse relationship between the predictive power of the SMM and robust parameter estimation. As n increases the predictive power of SMM increases at the expense of unreliable parameter estimation, which in turn may start to hurt the predictive power of the model. Following extensive experimentation, $2 \leq n \leq 6$ are found to work best for natural language models. In this work, for program behavior prediction we use $n = 8$ to have a fair comparison with prior work. However, we also provide an evaluation of SMM performance with smaller n. By restricting the conditioning information to the previous seven $(n-1)$ samples, we are making a simplifying assumption. Although samples further back in history potentially also have an influence on the identity of the next sample, higher order models provide diminishing returns due to the fact that the number of parameters in the SMM model is exponential in n. Converting the entire sequence of metric samples that are observed up to now to a set of finite sequence of n samples, allows us to compare any sequence with any other sequence in an efficient manner.

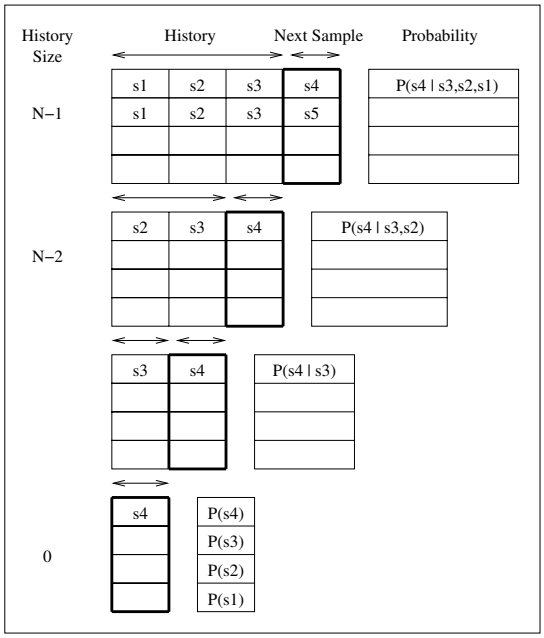


Figure 3. Description of the statistical metric model with back-off for $n = 4$.

4.3 Formulation of SMM

The SMM model is a probability distribution, $P(s)$, over L samples $S = s_1, s_2, ..., s_L$, that attempts to reflect the frequency with which each finite sequence $s = s_1, s_2, ..., s_l$ $(l < L)$ occurs in a metric.

$$\begin{aligned} P(s) &= P(s_1)P(s_2|s_1)...P(s_l|s_{l-1}...s_1) \\ &= \prod_{i=1}^{l} P(s_i|s_{i-1}, s_{i-2}, ..., s_1) \end{aligned} \quad (1)$$

In SMM, the probability of a string $P(s)$ is expressed as the product of the probabilities of the samples that compose the sequence, with each sample probability is conditional on the identity of the last $n-1$ samples. Without loss of generality, we can express the probability of a s, $P(s)$ as:

$$P(s) = \prod_{i=1}^{l} P(s_i|s_1^{i-1}) \approx \prod_{i=1}^{l} P(s_i|s_{i-n+1}^{i-1}) \quad (2)$$

where s_i^j denotes samples $s_i, ..., s_j$. In order to simplify the description and formulation of the SMM we consider the case $n = 2$. The extension of formulation and results to higher order models are trivial. By setting $n = 2$, we make the approximation that the probability of a sample only depends on the

identity of the immediately preceding sample, hence we can approximate $P(s)$ as:

$$P(s) = \prod_{i=1}^{l} P(s_i|s_{i-1}) \quad (3)$$

This probability distribution can be estimated with maximum likelihood estimation (MLE) technique:

$$P(s_i|s_{i-1}) = \frac{C(s_{i-1}, s_i)}{C(s_{i-1})} \quad (4)$$

where $C(x)$ denotes the number of times the sequence x occurs in the metric. This is called the maximum likelihood (ML) estimate for $P(s_i|s_{i-1})$.

4.4 Model Smoothing

While intuitive, the ML estimate has certain drawbacks due to data sparseness when the amount of metric data is small compared to the size of the built model. For example, for a model with $n = 8$, we can potentially have $20^8 = 25.6$ billion unique sequences of length 8. However, in practice, all applications combined exhibit less than 10K unique patterns, which is negligible compared to 25.6 billion. The conventional probability estimate for each unseen sequence would be zero. However, just because an event has never been observed so far does not mean that it cannot occur in the future. To overcome these shortcomings of the ML probability estimates, we apply "model smoothing" to the relative frequencies to make sure that each probability estimate is larger than zero.

Various smoothing techniques can be devised to ensure that the probability estimates are greater than zero for samples which do not occur in the training data.

A widely used set of smoothing methods is based on *absolute discounting*, which interpolates higher order n–gram models with lower order n–gram models. When there is insufficient data to estimate a probability in the higher order model, the lower order model can often provide useful information. The higher order distribution is created by subtracting a fixed discount $D < 1$ from each nonzero count. We use an interpolated version of the absolute discounting [7], which was shown to provide the best performance in natural language modeling applications. For finite metric sequences with nonzero counts, this distribution has the following general form:

$$\begin{aligned} P_{int}(s_i|s_{i-n+1}^{i-1}) &= \frac{C(s_{i-n+1}^{i}) - D}{\sum_{s_i} C(s_{i-n+1}^{i})} \\ &+ \alpha(s_{i-n+1}^{i-1}) P_{int}(s_i|s_{i-n+2}^{i-1}) \end{aligned} \quad (5)$$

where $P_{int}(s_i|s_{i-n+2}^{i-1})$ is the lower order smoothing distribution. Normalization constraints fix the value of $\alpha(s_{i-n+1}^{i-1})$:

$$\alpha(s_{i-n+1}^{i-1}) = D \frac{n_{1+}(*, s_{i-n+1}^{i-1})}{\sum_{s_i} C(s_{i-n+1}^{i})} \quad (6)$$

where $n_{1+}(*, s_{i-n+1}^{i-1})$ represents the number of bins for which $C(s_{i-n+1}^{i}) > 0$.

4.5 Implementation of SMM

While our SMM-based prediction approach is inspired from natural language modeling, there are unique differences in our implementation stemming from the different characteristics of computational flow in computing systems. First, typically, the natural language model is built from a previously collected existing corpus once, and the same model is used for speech recognition or information retrieval without any model update. Even though the dialog or topic context information could be used to further improve the prediction of the words to be uttered at any point in the text/conversation, the information contained in the existing corpus dominates the probability of which word is to be expected next. Second, in natural language a sentence contains a finite number of words. However, in workload behavior prediction, the entire sequence of execution is not observed when the SMM is used for prediction. In fact, in practice monitored system execution is a continuous process that produces a stream of workload characteristics without any particular beginning or end. The metric data that is of interest to the SMM predictor keeps coming from the beginning system startup to the next system halt, which can span days or months. Because of this, the SMM must be trained/updated constantly. The key question here is whether the computational complexity of model training/updating presents a challenge on the practicality of the SMM.

We can estimate the computational complexity of the SMM model training. For that, we need to know three parameters: i) number of quantization bins, ii) length of finite sequence (n), iii) how often we need to update the model. In our experiments, we have $V = 20$ quantization bins, $n = 8$ and we update the SMM parameters at every sample. For such a setting, the computational overhead for each model update would be bounded by $O(kVn)$ division and multiplication operations, where k is a small constant ($k \leq 3$). The storage requirements could be a more important concern for large n value and very long metric sequences.

On average the SMM model storage for model order 8 is around 10KB, which is in the same ballpark as the table-based predictor [1] As the metric data becomes longer the SMM model size will increase as well. However, this is not a concern from a practical implementation point of view, as there are a number of effective methods that can keep the model size under control without compromising the model performance [11]. Moreover, the model size can inherently be controlled by limiting the number of quantization bins and the model order. Fewer quantization bins and smaller model order will lead to smaller model size.

Based on the described computational upper bounds for the model update, the overall computational overhead of the SMM is constrained to be less than a thousand multiply and divide operations. This translates to compute time overheads on the order of microseconds. Therefore, the proposed schemes can be implemented within the operating system software in context switch time granularities with no visible performance impact.

[1] Each bin can be represented with a byte and each entry has model order of 8. For a 1024 entry table the total storage is $1024 \cdot 8 = 8KB$

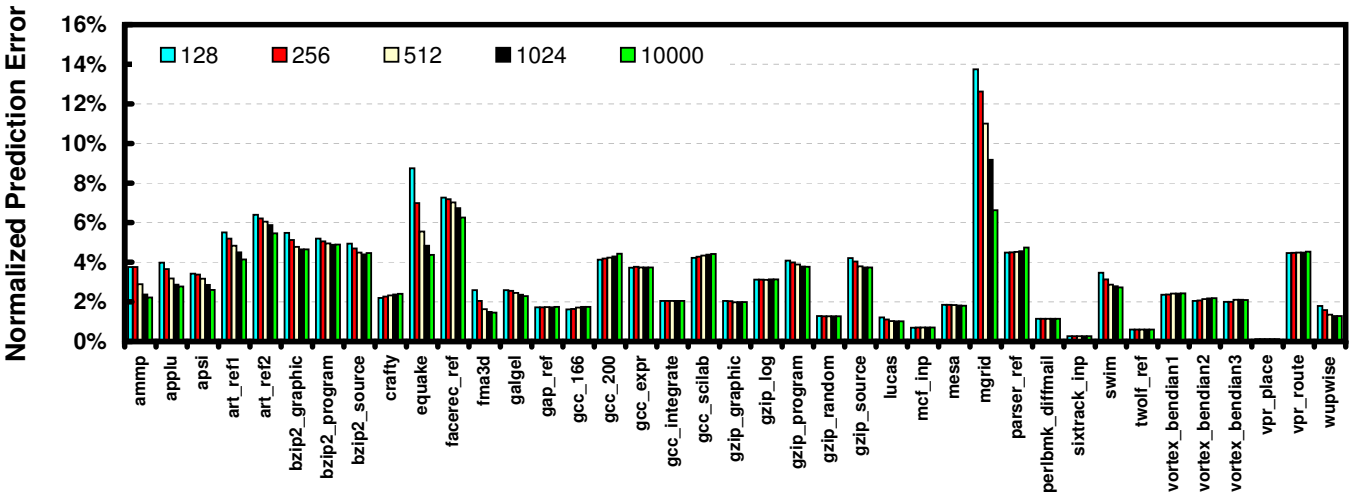


Figure 4. Normalized mean prediction error for IPC using table based predictor with various table sizes of 128, 256, 512, 1024, 10000.

4.6 Advantages of SMM

By expressing various short term and long term sequential phenomena in terms of simple parameters in a statistical model, SMM provides an easy way to deal with complex computer metric prediction. One of the important features of the SMM in comparison to the previous methods is its ability to model long term patterns. Last value predictor, table based predictor and others attempt to use short term dependencies, focusing on the previous $n-1$ samples to predict the next sample. This of course is useful but not sufficient to model long term dependencies where repetitive patterns go beyond the window of past n samples.

Another major advantage of the SMM is its ability to model patterns of variable length. We show in Figure 3 that SMM has the ability to match patterns of variable length m, where $(1 \leq m \leq n)$. This is a major issue with table based predictor, as its patterns are fixed in length. SMM takes a pattern of length n and checks whether there is such an entry in the model, if so, it uses the probability for that entry. If not, then it checks models of lower order. For a given history, the predicted next sample is selected based on its probability.

5 Methodology

We perform the experiments on an IBM POWER4 [23] server platform with the AIX5L operating system. The results present per-thread behavior running in multi-user mode on a lightly loaded machine. We use the hardware performance counters, with a sampling tool that works on top of the AIX Performance Monitoring API (PMAPI). The sampler ties together counter values to a particular thread, including all library calls and system calls performed by that thread. The prediction is performed at 10 ms time scales. All the experiments are performed with the SPEC CPU2000 suite using reference datasets. All benchmarks are compiled with XLC and XLF90 compilers with the base compiler flags. The data in the experimental sections shows prediction performance with the instruction per cycle (IPC) metric. However, the same prediction approach can be applied to other architectural metrics.

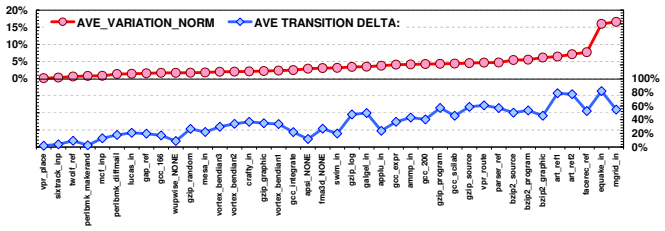


Figure 5. Variation in benchmarks.

The results shown are based on a quantization level of 20 bins. This represents the high end of the experimented phase granularities, limiting within-bin quantization error range to 5%.

6 Experimental Results

We run a rigorous and extensive set of experiments to compare SMM, table based and last value predictors. We also investigate SMM predictor in-depth for its performance in relation to various parameters. In order to provide a fair comparison, we also perform a sensitivity analysis for the table based predictor using different table sizes. Figure 4 presents the normalized mean prediction errors for different table sizes. Our results confirm the previous studies [16] in that using table sizes larger than 1024 does not provide significantly different prediction results. Therefore, in our experiments we use a table based predictor with a fixed size of 1024 entries.

6.1 Metric Variation

The value of a good prediction scheme is realized when the predicted workload characteristics show significant variability. In other words, if the workloads of interest show little temporal variation, it is an overkill to design anything beyond a trivial predictor such as the last value predictor. However, for benchmarks with rapidly changing characteristics, such a predictor performs unfavorably. Therefore, it is important to understand the variability and "predictability" of different workloads used in predictor evaluation.

Figure 5 shows the available variation in the workloads used in this work under two criteria. The upper plot shows

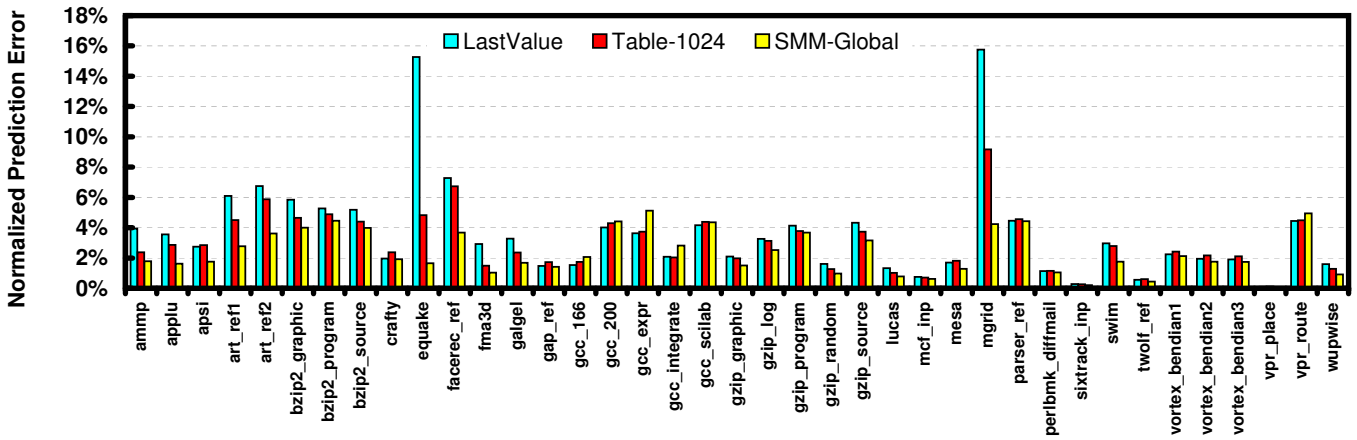


Figure 6. Improvements with the SMM predictor, compared to last-value and table-based predictors.

the average *sample-to-sample* variation in the tracked metrics. normalized to the overall dynamic range of the workload. In the lower plot, the magnitude of variation is decoupled from the *occurrence* of a variation by profiling how often two consecutive samples belong to different phases. This measure is useful to understand the deviation of each workload from a purely *flat* workload. In the figure, the benchmarks are sorted in increasing variability. Thus, workloads towards the right end exhibit the highest variability with the last six workloads showing more than 5% sample-to-sample variation.

6.2 Metric Tracking Using SMM Predictor

The optimal model order n depends on the specific benchmark and available data to train the SMM. In this study, we set $n = 8$, simply because the table based predictor, which is considered as one of the baseline methods uses a sequence length of 8. We also use the last value predictor both as a baseline to compare and as a backoff predictor for the table based predictor. Using larger model orders can allow us modeling longer patterns but the underlying model parameters may not be robustly estimated. However, using smaller model orders may not have enough predictive power to model any patterns embedded in the metric sequence. We believe that $n = 8$ is a reasonable compromise between these two competing goals. In later sections, we also look at this tradeoff for different values of n.

Figure 6 shows a comparison of last value, table based and SMM predictors across all benchmarks. The SMM predictor improves prediction errors by 19% on average over all the experimented workloads compared to the table based predictor. This improvement is even further emphasized, with 43% relative reduction in prediction error for the top five highly-varying applications, namely as `mgrid`, `equake`, `facerec`, `art_ref1`, `art_ref2`. In comparison to the last value predictor, SMM improves prediction errors on top five highly-varying applications by 63%. The largest improvements of about 10–fold (15.3% vs. 1.6%) and 3–fold (4.8% vs. 1.6%) are achieved compared to the last value and table based predictors, respectively on the `equake` benchmark.

Figure 7 shows how the prediction performance of the different prediction approaches change over time, with the `equake` benchmark. In the first panel the entire IPC sequence

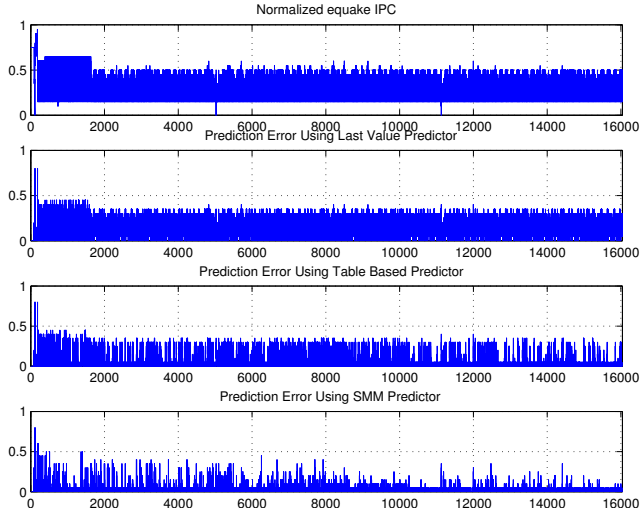


Figure 7. Normalized IPC prediction errors for different predictors over the quantized samples.

of `equake`, normalized to its max, is plotted. In the other panels, the prediction errors are plotted for the last value, table based and SMM predictors. We clearly observe from the plots that the table based predictor outperforms the last value predictor and the SMM predictor outperforms both predictors. As expected, as the amount of data increases the SMM parameter estimation becomes more reliable. As a result the SMM prediction performance improves towards the end of the figure. Online learning and adaptation are two critical features that an adaptive predictor must have. SMM has the ability to learn and adapt itself constantly. As it learns and adapts itself to the changing phase behavior, the prediction accuracy improves. On the other hand, in the same figure, the errors for last value predictor appears to be evenly distributed across the time scale. This is expected, as this simple predictor does not employ any adaptation based on previously-observed application behavior. The table-based predictor's performance also improves slightly with increasing amount of data, since it also aims to discern the patterns in application behavior. However, the improvements in the prediction accuracy do not come close to those of the SMM predictor, as the SMM predictor can successfully leverage both long-term and varying-duration ap-

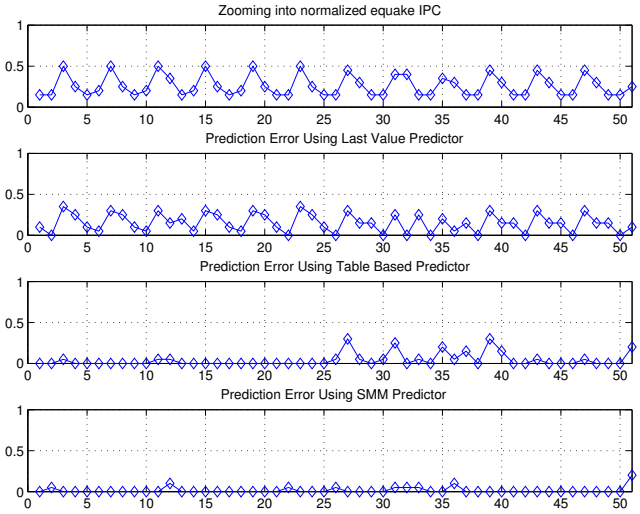


Figure 8. Normalized IPC prediction errors for different predictors over an execution segment.

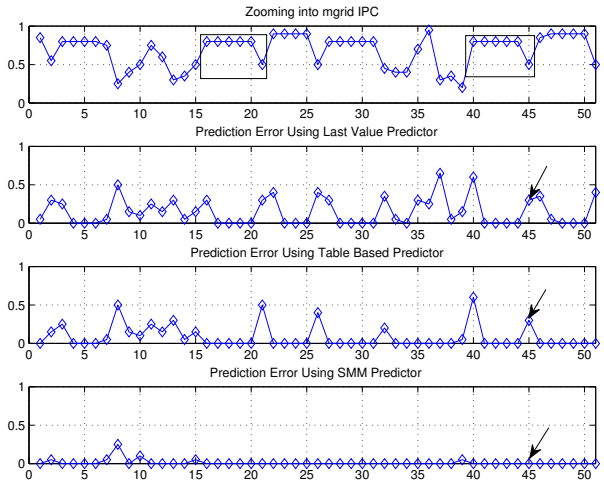


Figure 9. Benefit of backing off to lower order models for SMM.

plication characteristics.

We also zoom into a segment of the `equake` data to provide more concrete insights on how different predictors are performing. In the first panel of Figure 8, we plot a segment of the normalized `equake` IPC data. This segment of the data is somewhat periodic with significant differences in values between consecutive samples. In the second panel, the corresponding normalized prediction error is plotted using the last value predictor, where the errors are almost on the same scale as the data samples. In the third panel table-based predictor errors are displayed. The table-based predictor models the periodicity in the data to some extent, as seen in the beginning and end of the plot. However, each time there is a slight variation in the observed behavior, the fixed-pattern-based approach fails and the predictor backs off to the last value predictor as seen between samples 25 and 40. However, SMM is very robust to small variations in the patterns and is able to provide accurate prediction.

Figure 9 underlines one of the main strengths of SMM as compared to the table based predictor. That is, SMM backs off to lower order models when the maximum sequence match is not found. In the figure, we highlight two patterns with boxes, which contain identical samples of length 6. Even though, the first 5 samples are the same in both boxes, table based predictor does not have any match simply because it stores samples of length 8. When there is no match, then it backs off to the last value predictor, which does not produce good prediction, as pointed out by the arrows. However, SMM backs off to lower order models and finds the matching pattern of length 6, which is observed in the first box, and uses it to perform accurate prediction in the second box. This is of course the case, given the past 5 samples in the box, the next sample 10 has the highest probability among all the possible outputs (1,...,20). SMM prediction considers the conditional (on the history) probability of the each possible metric output and picks the one that has the highest probability.

The other main strength of the SMM predictor is its ability to model long range patterns, which improves the prediction. In order to underline this feature of SMM, we extend the runtime duration of the experimented benchmarks by a factor of 2 (shown as x2 in Figure 10) by concatenating workloads back to back. With this approach we provide the predictors with runtime histories which are twice as long as the original runs. In Figure 10, we plot the normalized mean prediction errors for the original benchmark runs and the extended benchmark runs. We also plot the results for the table based predictor in both cases. Not surprisingly, the prediction performance of the table based predictor shows almost no improvement except for two benchmarks. Those two benchmarks are `gcc_exp` and `gcc_integrate`, which are both very short in metric sample size. SMM, on the other hand, provides significant improvements for almost all the benchmarks. The only exception is `swim`, where we observe a slight increase in prediction errors of both predictors. It is not surprising to see that SMM performance improves on the extended-duration benchmarks, since SMM has a memory of observing all the prior characteristics before. SMM starts to improve the prediction in the second half of the duplicated benchmark data. The average improvement across all benchmarks is 15% for the SMM predictor, whereas it is only 3% for the table based predictor. This shows the substantial benefits of the SMM predictor in the longer term.

It is important to emphasize that the ability of the SMM predictor to capture long-term patterns is not directly proportional to its chosen model order/history size, or its quantization bins. The key strength of the SMM predictor stems from a fundamental difference from the existing predictors. While pattern-based or statistical predictors rely on recent history and discard old pattern information in favor of new observations, SMM actually eliminates this temporal dependence by relying on its probability-based history information. Thus, older but dominant patterns survive in SMM model, and provide significant influence in future predictions. SMM brings out two unique features: i) it can model patterns of varying length, ii)

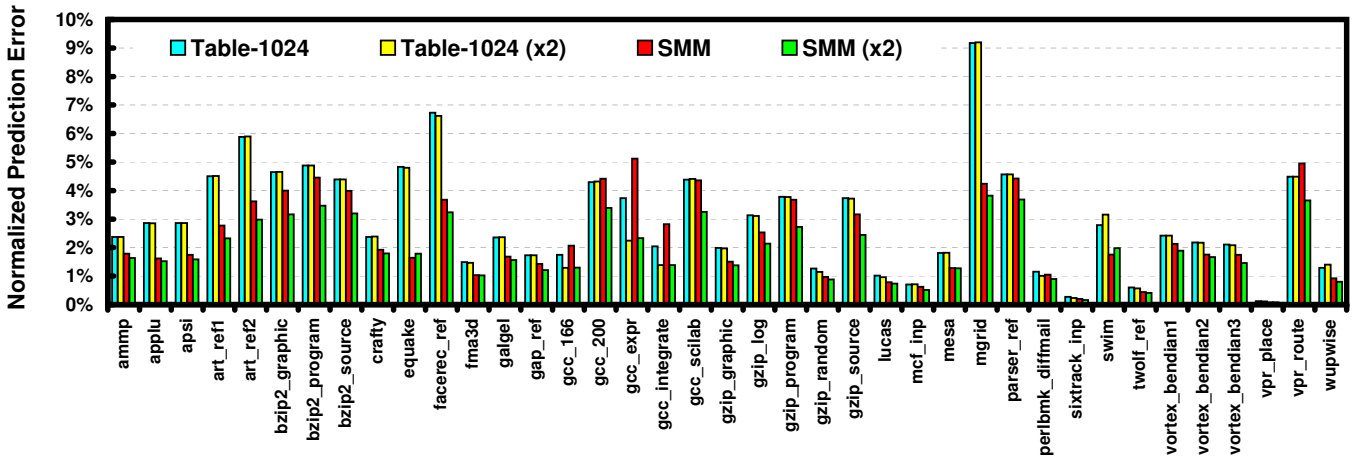


Figure 10. Improvement in SMM predictor accuracy with longer benchmark execution times.

it can model long range patterns by tracking observed patterns with their respective probabilities. When capacity requirements dictate discarding some patterns, this pruning is done not by temporal proximity, but with respect to the strength of the patterns as indicated in the SMM pattern tables. This leads to a much higher accuracy in predicting future behavior.

6.3 Impact of Model Order on SMM Predictor Performance

For SMM, model order (n) and the history size are related via $HistorySize = n - 1$. We mentioned that the performance of SMM predictor largely depends on the model order and the available data to estimate the model parameters. Smaller model order limits the predictive power of the SMM predictor. However, using larger model orders may adversely affect the parameter estimation step - which in turn may degrade the prediction performance. We are not aware of a "grand" recipe that would tell us the optimal model order given the amount of training data. However, one should keep in mind that even though the metric data sizes for specific applications are fixed, in reality SMM is designed to operate continuously—for days or even months—during system uptime, with a very large set of running metric samples. In that case using larger model order can benefit the prediction performance.

In this section, we implement three SMM versions with different model orders and observe their effect on the prediction performance. Each benchmark can be described better with a specific model order and may achieve the smallest prediction accuracy. In Figure 11, we provide prediction errors for all benchmarks with model orders: $\{4,6,8\}$. In general, as the model order increases, overall prediction accuracy also increases leading to lower prediction errors. Typically, models with larger model order are good in describing the fine structure in the segment of the data that is observed so far. However, in the absence of large data, this comes at the expense of poor generalization for the unseen future observations, which are to be predicted. There are few exceptions to this overall trend. For example, model order 4 (n=4) for parser_ref achieves a slightly lower prediction error than model order 8 (n=8). We also observe that the improvements with the larger model order appears to be leveling off for $n \geq 6$ for most of the benchmarks with the exception of `mgrid`, where there is further possible reduction in prediction error with larger model sizes.

6.4 Application of SMM to Power Management

This section explores how the predictions provided by the SMM predictor can be applied to dynamic power management. For this evaluation we use SMM predictor to predict the memory access rate of applications, and use these to control dynamic voltage and frequency scaling (DVFS). Such workload-behavior-based DVFS is well studied in prior work [24, 25]. These show that the memory access rates of applications are a strong indicator of the appropriate DVFS state an application can run with limited performance degradation. Applications with higher amount of memory accesses exhibit higher potential for running at lower frequencies as the impact of running the processor slower has much less performance impact for these applications.

We profile the memory access rates for all applications. Then, based on the observed memory behavior, we categorize different execution characteristics into different representative bins. We reference actual power and performance measurements collected at different DVFS states [13] to define the bin boundaries and to characterize the power-performance tradeoffs of running each representative bin at different DVFS settings. We use memory access rates as the main differentiator for categorizing execution into bins. In our evaluation system, we consider eight bins, corresponding to eight DVFS states. That is, when a predictor predicts the next application phase as bin_i the processor is proactively set to DVFS setting i.

Figures 12(a) and 12(b) show the power-performance behavior of running the processor at different DVFS states during different execution characteristics. Here, $bin1$ corresponds to an execution region with minimal memory accesses and $DVFSstate1$ represents the highest DVFS setting. When the execution behavior is similar to $bin1$ running the application in DVFS states other than 1 result in significant performance degradation. In contrast, using a higher DVFS state for an execution region with higher memory accesses—i.e., higher bin number—is much more beneficial due to the smaller performance degradation impact. While running such a region at a lower DVFS setting still improves performance slightly, there

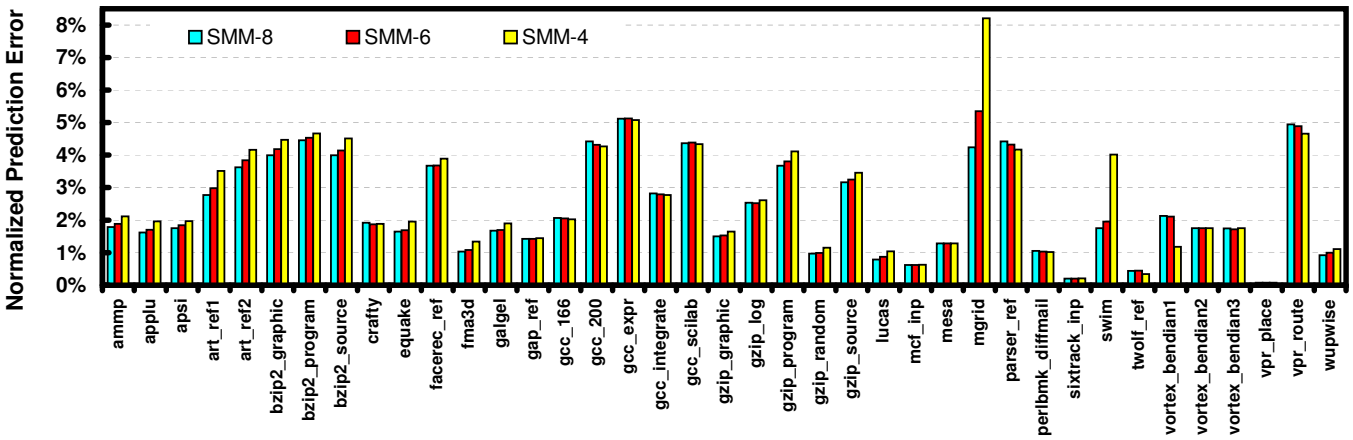
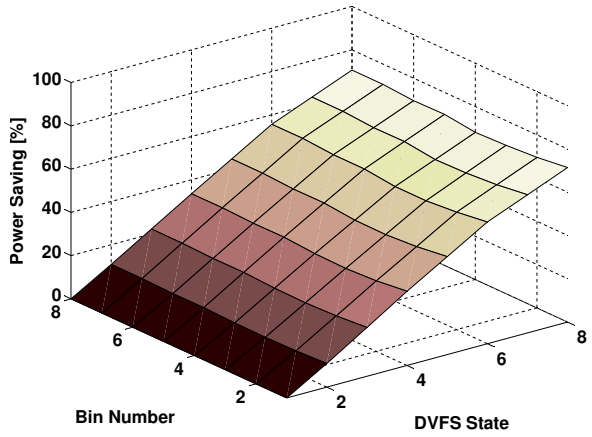


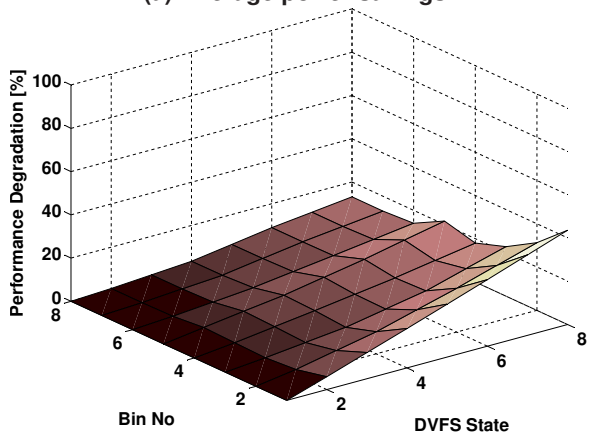
Figure 11. SMM prediction performance for different model orders of 4, 6, and 8.

is significant lost power savings opportunity.

In our evaluations, we consider the three predictors: last-value predictor, table-based predictor and our SMM predictor. Similar to the prior experiments, we continuously predict application behavior at fixed sampling intervals. Based on the predicted execution behavior, i.e., memory access rate, we set the corresponding DVFS state for the following period. Then, at the end of this period we observe the achieved power savings and the associated performance degradation based on the actual execution behavior in this past interval.



(a) Average power savings.



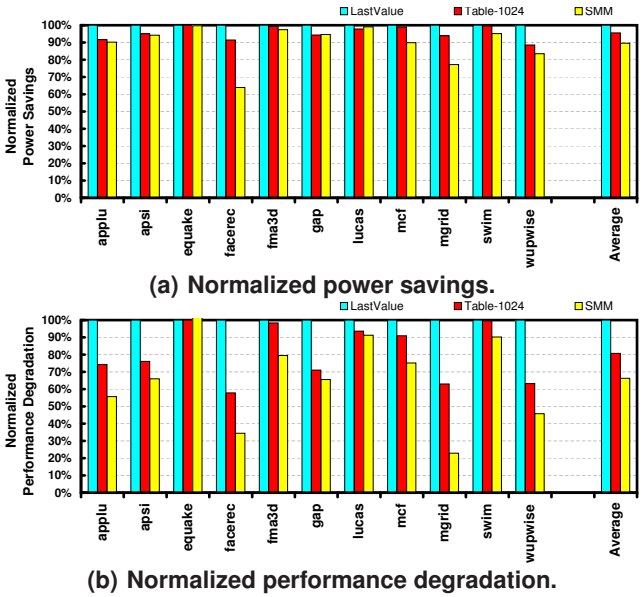
(b) Average performance degradation.

Figure 12. Power savings and performance degradation at different DVFS states for different bins.

We evaluate all three predictors for all applications. We accumulate the overall power savings and performance degradation throughout the execution of the applications and determine the average power savings and the experienced performance degradation for each application. Figures 13(a) and 13(b) depict these results. While we have included all applications in our experiments, in the figures we only show a subset of these. This is because, all the excluded benchmarks exhibit either very low variability or they are highly CPU-bound and therefore do not benefit from our employed DVFS-based power management. All predictors perform similarly for these applications.

In Figures 13(a) and 13(b), we depict the normalized power savings and performance degradations relative to the last-value predictor. Thus, the figures show the relative effectiveness of the different predictors. Overall, among the three predictors, the last-value predictor achieves somewhat higher power savings, followed by the table-based predictor. However, as Figure 13(b) shows, these power savings are achieved at the expense of different levels of performance degradation. Here the distinction among the three predictors is much more significant, where the table-based predictor performs better than the last-value predictor and the SMM predictor significantly outperforms both predictors. This outlines the benefit of the SMM predictor's higher prediction accuracy. While the SMM predictor achieves slightly lower power savings, with less than 10% difference compared to last-value and less than 5% compared to the table-based predictor, it significantly reduces the performance impact. The SMM predictor reduces overall performance degradation by 34% compared to the last-value predictor and by 19% compared to the table-based predictor.

These results highlight the potential benefit of employing our SMM predictor for dynamic management of computing systems. While the use case shown here pertains to a specific architectural management technique, the SMM predictor in general can be employed in other systems management techniques that benefit from the accurate prediction of



(a) Normalized power savings.

(b) Normalized performance degradation.

Figure 13. Power savings and performance degradation achieved by the three prediction methods.

dynamically-varying application characteristics for proactive dynamic adaptations.

7 Conclusions

We describe a new method called Statistical Metric Model (SMM) for predicting dynamically-varying program behavior. SMM is a probabilistic model that learns application characteristics at runtime and captures long term, dominant application behavior. There are four main strengths of SMM compared to existing predictors. First, it models long term global patterns in application behavior. Second, the predictor can respond to variable-length patterns. Third, it is resilient to small fluctuations in the observed patterns. Last, the SMM predictor has the ability to adapt itself; as it learns more it predicts better. We present a series of experiments that demonstrates these strengths of the SMM predictor, as well as its superior accuracy. These studies show that the SMM predictor reduces prediction errors by up to 10X and 3X compared to the last value and table based predictors respectively, with an average improvement of more than 60% and 40% for highly varying benchmarks. We also show the application of the SMM predictor to dynamic power management, where the better prediction accuracy of the SMM predictor achieves superior power-performance trade-offs compared to the other predictors.

References

[1] D. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster. Dynamically Tuning Processor Resources with Adaptive Processing. *IEEE Computer*, 36(12):43–51, 2003.

[2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law Through EPI Throttling. In *Proc. of the 32nd Inter. Symposium on Computer Architecture (ISCA-32)*, 2005.

[3] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Inter. Symposium on Microarchitecture*, pages 245–257, 2000.

[4] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-Driven Energy Accounting for Dynamic Thermal Management. In *Proc. of the Workshop on Compilers and Operating Systems for Low Power (COLP'03), New Orleans*, Sept. 2003.

[5] W. L. Bircher, M. Valluri, J. Law, and L. K. John. Runtime identification of microprocessor energy saving opportunities. In *Proc. of the 2005 Inter. Symposium on Low Power Electronics and Design (ISLPED)*, 2005.

[6] P.E. Brown, S.A. Della Pietra, V.J. Della Pietra, J.C. Lai and R.L. Mercer. An Estimate of an Upper Bound for the Entropy of English. In *Computational Linguistics*, 18(1):31–40, March 1992.

[7] S. Chen and J. Goodman. An Empirical Study of Smoothing Techniques for Language Modeling. In *Technical Report, TR-10-98, Computer Science Group, Harvard University*, 1998.

[8] K. Choi, R. Soma, and M. Pedram. Dynamic Voltage and Frequency Scaling based on Workload Decomposition. In *Proc. of Inter. Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2004.

[9] A. Dhodapkar and J. Smith. Managing multi-configurable hardware via dynamic working set analysis. In 29th Annual Inter. Symposium on Computer Architecture, 2002.

[10] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *IEEE PACT*, pages 220–231, 2003.

[11] J. Goodman. Language Model Size Reduction by Pruning and Clustering. In *Inter. Conference on Spoken Language Processing*, Beijing China, 2000.

[12] M. Huang, J. Renau, and J. Torrellas. Positional Adaptation of Processors: Application to Energy Reduction. In *Proc. of the Inter. Symp. on Computer Architecture*, 2003.

[13] C. Isci, G. Contreras, and M. Martonosi. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *Proc. of the 39th ACM/IEEE International Symposium on Microarchitecture (MICRO-39)*, 2006.

[14] C. Isci and M. Martonosi. Identifying Program Power Phase Behavior using Power Vectors. In *Proc. of the IEEE Inter. Workshop on Workload Characterization (WWC-6)*, 2003.

[15] C. Isci and M. Martonosi. Detecting Recurrent Phase Behavior under Real-System Variability. In *Proc. of the IEEE Inter. Symposium on Workload Characterization*, Oct. 2005.

[16] C. Isci, M. Martonosi, and A. Buyuktosunoglu. Long-term Workload Phases: Duration Predictions and Applications to DVFS. *IEEE Micro: Special Issue on Energy Efficient Design*, 25(5):39–51, Sep/Oct 2005.

[17] F. Jelinek and R.L. Mercer. Interpolated estimation of Markov source parameters from sparse data. In *Pattern Recognition in Practice, E. S. Gelsema and L. N. Kanal, Eds. Amsterdam: North-Holland,* 1980.

[18] J. Lau, S. Schoenmackers, and B. Calder. Transition Phase Classification and Prediction. In *11th Inter. Symposium on High Performance Computer Architecture*, 2005.

[19] R. Sarikaya and A. Buyuktosunoglu. Predicting program behavior based on objective function minimization. In *Proc. of the IEEE Inter. Symposium on Workload Characterization (IISWC)*, Sept. 2007.

[20] X. Shen, Y. Zhong, and C. Ding. Locality Phase Prediction. In *11th Inter. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Oct. 2004.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In 10th Inter. Conference on Architectural Support for Programming Languages and Operating Systems, Oct 2002.

[22] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proc. of the 28th International Symposium on Computer Architecture (ISCA-30)*, June 2003.

[23] J.M. Tendler, J.S. Dodson, S. Fields, H. Le and B. Sinharoy. POWER4 system microarchitecture. In *IBM Journal of Res. and Dev.*, 46(1), 2002.

[24] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proc. of the Inter. Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002), Grenoble, France,*, Aug. 2002.

[25] Q. Wu, V. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D. W. Clark. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *Proceedings of the 38th International Symp. on Microarchitecture*, 2005.

[26] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proc. of the 11th Inter. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, 2004.

Analyzing and Scaling Parallelism for Network Routing Protocols

Abhishek Dhanotia, Sabina Grover, Greg Byrd
Department of Electrical and Computer Engineering
North Carolina State University, Raleigh, NC, USA
(adhanot, sgrover, gbyrd)@ncsu.edu

Abstract: **The serial nature of legacy code in routing protocol implementations has inhibited a shift to multicore processing in the control plane, even though there is much inherent parallelism. In this paper, we investigate the use of multicore as the compute platform for routing applications using BGP, the ubiquitous protocol for routing in the Internet backbone, as a representative application.**

We develop a scalable multithreaded implementation for BGP and evaluate its performance on several multicore configurations using a fully configurable multicore simulation environment. We implement several optimizations at the software and architecture levels, achieving a speedup of 6.5 times over the sequential implementation, which translates to a throughput of ~170K updates per second. Subsequently, we propose a generic architecture and parallelization methodology which can be applied to all routing protocol implementations to achieve significant performance improvement.

I. INTRODUCTION

The past decade (1999-2009) has witnessed a tremendous increase in the size and scope of the Internet [1]. With the increase in user base and the bandwidth demand per user, the network backbone has scaled tremendously. Routers are the main constituent of a network backbone and perform most of the intelligent functionality of the network. A router has two primary software components: Control Plane and Data Plane. Data plane functionality involves forwarding traffic from an incoming interface to an outgoing interface. Control plane applications, on the other hand, primarily maintain information about all the nodes in the internet. Routing protocols form a critical part of the network control plane and help in determining which outgoing interface the packets should take if they are destined for a particular network or IP address. In order to determine the correct outgoing interface, routers maintain reachability information, usually in the form of routing tables.

Border Gateway Protocol (BGP) [2] is the de facto standard for routing in the internet core. Since the Internet backbone consisting of Autonomous Systems (AS) performs a major chunk of data forwarding and is based on routing decisions made by the BGP protocol, it makes studying BGP all the more important. With the increased usage of the Internet, the data plane has scaled well with a lot of new hardware and architecture solutions being proposed for increasing data forwarding bandwidth and throughput [3][4]. However, there has not been much work on scaling control plane applications and architectures.

The number of BGP routers and the routing table sizes within each router has grown dramatically, thus requiring higher amounts of compute resources. During peak activity, the number of update messages processed by routers has been known to grow exponentially [5] compared to the updates that a router generally processes. The latency of a single update packet is also important, because it affects convergence time – the delay involved in propagating a routing change across the network. Large convergence time can lead to lost traffic, routing loops, and black holes [5]. Increasing the processing power of the routers is the obvious solution for these problems.

Multicore processors are known to give great performance benefits for applications with high inherent data level parallelism. Several router data plane applications have already been ported to multicore machines in order to achieve scalability for high traffic and bandwidth requirements. In this study, we show that control plane applications also tend to exhibit significant data level parallelism which makes multicore processors an ideal choice of compute platform for these applications.

Running any application on multicore requires it to be written in a parallel fashion, so as to exploit the available processing power of the underlying machine. Most contemporary control plane applications are written sequentially, with almost no support for concurrent execution. The scale and complexity of control plane applications makes porting them to multicore a challenging task. As most routing protocol suites carry over legacy code, and only incremental development is done when new features are introduced in the protocol, there is a lot of inertia to overcome when changing their software architecture to add support for multithreading. Hence, programmability of routing protocols for multicore processors is another important area where research needs to be done. Schemes and methodologies for multithreading the router software which require minimal changes to the software architecture would greatly help in speeding up the process of migrating the routing applications to multicore processors.

In this work, we show the benefits of using multicore processors in network routing protocols, taking BGP as a representative application. We also propose a task partitioning

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

scheme for converting the legacy serial routing protocol code into a multithreaded one, point out bottlenecks in the system, and suggest ways to mitigate them. A detailed architectural analysis is done by studying the performance of this scheme for different parallel architecture paradigms including private *vs.* shared L2 cache, different cache sizes and coherence mechanisms. We also evaluate the interconnect traffic behavior by studying the link utilization on different topologies.

Although, we do all our analysis using BGP, applying similar techniques to other inter-domain and intra-domain routing protocols, for instance OSPF and RIP, should give similar performance improvements.

The major contributions of our work are listed below;

- A comprehensive analysis of the generic software architecture for routing protocols and several avenues to extract parallelism. We identify bottlenecks in the parallel implementation and suggest ways to mitigate them.
- A detailed analysis of the underlying architecture to find an optimized configuration.
- A realistic BGP simulation environment which enables performance analysis at both the software and hardware architecture levels.
- A task partitioning scheme that gives consistently good performance irrespective of the load across various peers.

II. SEQUENTIAL BGP IMPLEMENTATION

This section presents the basic details about BGP implementation and describes the software architecture of Quagga routing suite [6] which is used as the baseline code and converted to a parallel application. Quagga is an open source routing protocol suite supporting several intra and inter domain routing protocols. Quagga BGP protocol stack is implemented as a completely sequential code with no support for multithreading or performing different tasks in parallel.

A. BGP State Machine

Any routing suite supporting BGP has to maintain the standard BGP state machine which is defined in its RFC [2]. BGP is a path vector protocol where neighboring nodes (called BGP peers) exchange reachability information about all the IP addresses they know of. A routing table is maintained called Routing Information Base (RIB), which stores paths to all the IP addresses which could be reached by this router. In order to exchange messages with the peers, a TCP connection is maintained with each active peer. The state machines are run for each peer and are independent of each other. The state transitions for one peer session do not affect those in the other session.

B. Packet Processing

The BGP router initially starts in passive open mode listening for any connections. When a peer comes up, the initialization process happens with a sequence of BGP Open and KeepAlive messages being exchanged. After the initialization is complete the connection is moved to established state and periodic keepalive messages are exchanged between the peers. Once in established state, when a BGP node receives update messages from the corresponding peer, it would make appropriate entries in the routing tables and advertise routes to its peers based on the policies it exchanged with the peer during the initialization phase.

C. Common Libraries

There are some common libraries which are used by all the routing protocols in the Quagga routing suite. They are maintained as a separate library base. The prominent ones among them are thread, stream and queue libraries.

- Threads in Quagga sequential code refer to the functions that need to be performed on any state transitions or timer expiry. Each thread holds a pointer to a function which gets executed when the thread is called. This is different from the threads that we talk about in our multithreaded implementation. From here on, the library threads would be referred to as tasks.
- Streams are byte buffers that are maintained for each peer session and hold the incoming and outgoing packets. Two streams are maintained, one for the incoming packets from a peer session and the other for outgoing packets.

D. Task Scheduling

The implementation maintains five different task queues: event, ready, read, write and background. Tasks to be performed by BGP router are segregated into these queues. Read queue holds tasks scheduled to read data from the incoming TCP buffer. When there is data to be read, the task is moved from the read queue to the ready queue. Similar is the case for write queue. All BGP timers are held in the background queue and moved to ready queue when they expire. Event queue holds all the events generated by the state machine. Tasks are picked up from the event and ready queues and are executed one at a time. Event queue has priority over the ready queue. Figure 1 shows the flow of tasks between different queues in Quagga.

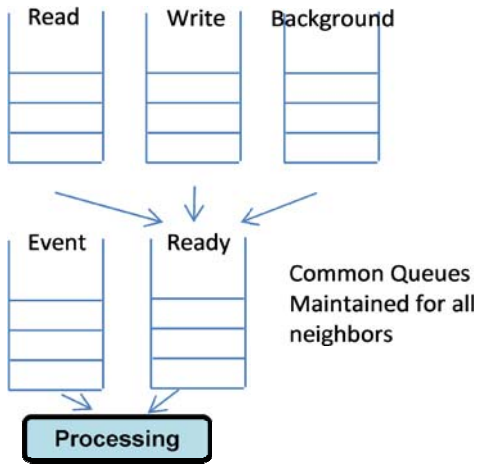


Figure 1: Task Scheduling in Quagga

III. PARALLELIZING BGP ROUTING SOFTWARE

This section presents the synchronization mechanism used in the parallel implementation and discusses two schemes for multithreading the Quagga BGP implementation namely Peer Based Task Scheduling and Dynamic Task Scheduling. Peer Based Task Scheduling (PBTS) scheme leverages on the fact that data from all incoming peers is independent of each other. Dynamic Task Scheduling (DTS) on the other hand follows a more load balanced approach by picking up tasks irrespective of the peer they are associated with.

We base our multithreaded implementation on Quagga software but our literature survey reveals that other prominent routing suites include XORP [7] and Cisco IOS have similar architectures. Hence, the parallelization techniques and optimizations that we propose for Quagga should be applicable to the other implementations as well.

A. Synchronization

Different lock variables are maintained in the multithreaded code to prevent contention among the threads for accessing key data structures. Primarily, there are three different types of lock variables that are maintained to prevent simultaneous access on the routing tables, peer specific byte buffers and global file descriptor sets respectively. Routing tables in Quagga are maintained as binary trees with each prefix (IP address) represented as a node in the tree. Routing table lock variables are maintained on a per node basis and enable maintaining a fine grain locking in the routing tables allowing multiple threads to modify the tree as long as they are accessing different locations in the tree. Peer Locks are maintained for each peer which controls access to the input and output streams of the peer. File Descriptor Set (FD Set) Lock ensures that when select logic is called for reading data from the TCP buffer, no other threads is accessing the FD set.

B. Parallelization Approach

The sequential implementation described in the previous section involves reading tasks from the head of event and ready queues and executing the associated functions one at a time. When executing the functions, more tasks are enqueued in the event and ready queues and the execution continues. A disadvantage of this approach is that all the tasks are executed sequentially even when they may be completely unrelated to each other. For instance, if two update messages are enqueued in the ready queue, they would be picked up one at a time and executed even when they are completely independent and can be processed simultaneously if there are multiple threads.

In a multithreaded implementation, there can be several ways in which tasks could be picked up from the ready and event queues and executed. An obvious approach could be to assign a peer to each thread and segregate tasks based on the peer to which they belong. This approach has been implemented in Peer Based Task Scheduling Scheme. Another approach is to spawn threads statically when the router starts and have each thread poll the ready and event queues and pick up executable tasks at the top of the queue. This approach has been implemented in Dynamic Task Scheduling Scheme.

C. Peer Based Task Scheduling

In Peer Based Task Scheduling scheme (PBTS), threads are spawned dynamically. A thread is maintained for each peer the simulated BGP node is connected with. Whenever the BGP node receives connection request from a peer, a new thread is spawned. This thread maintains a TCP connection with the respective peer and runs the corresponding BGP state machine. Each thread maintains separate read, write, event, ready and background queues. When processing incoming and outgoing updates, it acquires lock on the routing tables so as to update them in a critical section. A master thread common among all peers is also maintained, which handles all the housekeeping functions that are not associated with any peer. These include periodically clearing routing tables, checking socket buffers for new data and maintaining file descriptor sets etc.

D. Dynamic Task Scheduling

A limitation of PBTS scheme is that it is not scalable with the number of processors. The number of threads spawned is bound by the number of peers which is constant. Dynamic Task Scheduling (DTS) tries to overcome this problem by making the number of threads independent of the number of peers.

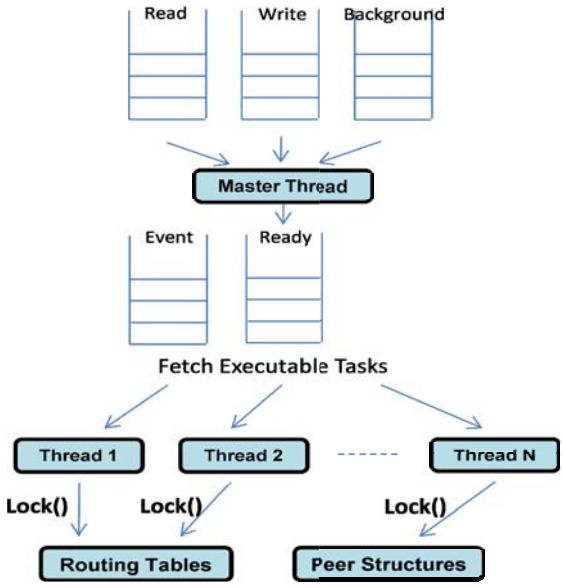


Figure 2: Dynamic Task Scheduling

In DTS, a pre-determined number of threads are spawned at the start of the router software. Common ready and event queues are maintained as in the case of original code. Each thread scans for executable tasks in the ready and event queues and processes them irrespective of the peer which they belong to. A master thread is responsible for moving data from read,

write and background queue to the ready queue. All the other threads just keep on scanning the ready and event queues and execute tasks as and when they arrive. This scheme thus decouples the number of threads from the number of peers and hence is more scalable. However, this scalability comes at a cost. Now that multiple threads could be operating on packets for the same peer, another level of synchronization needs to be maintained in order to keep the data coherent for a particular peer. As mentioned earlier, two streams are maintained for each peer to store the incoming and outgoing packets. When multiple threads operate the same peer, these streams could potentially get corrupted and hence need to be accessed in a critical section. A peer lock is maintained inside the peer structure which restricts the peer data structures to be modified by a single thread at a time. Figure 2 depicts the DTS scheme. A rare scenario of out-of-order prefix processing can occur if the same peer advertises the same prefix again and the second advertisement is processed before the first one. As packets are assigned to threads in order, it is easy to tell whether there's a thread already working on this same peer-prefix pair and squash its results. Since this would require very few comparisons it can be expected to have a negligible performance penalty.

IV. EVALUATION PLATFORM

This section describes the GEMS [8] and Simics [9] based environment that we use for our simulations. Subsection A describes the methodology used to generate realistic BGP traces. Subsection B presents how these traces are used to run full system simulations on Simics virtual system simulator and GEMS Memory simulator.

A. Generating BGP Traces

Current BGP deployment on the internet backbone connects 18,000 autonomous systems [10]. Generating representative traces similar in scale to the actual internet is a primary requirement for creating a realistic simulation environment. The methodology that we follow is to run a BGP network simulation having 1000 nodes communicating with each other, and then dump all packets exchanged by a single node with its peers. This node would then serve as the daemon node which we would run on a multicore machine and analyze its performance. Following are the steps that are followed in order to generate the packet traces

- Determine a network topology - BRITE Topology Generator [11] is used for generating the network topology which is used for the BGP network simulation

- BGP agent – NS2 does not have an inbuilt protocol simulator for BGP. BGP++ [12] is used as an NS2 agent that performs BGP simulation based on the given configuration. It is based on the same code used in Quagga routing software with modifications done in order to use the TCP/IP stack of NS2 for packet transmission and reception. BGP++ takes a configuration file as input which contains the peer information and the prefixes to be advertised by a particular node, and generates a log for all the state machine and path information. It also generates a dump (in MRT [13] format) for all the packets exchanged. This tool was used to perform network simulation for several BGP nodes and generate corresponding packet dump files.

- BGP++ Configurator - BGP++ configurator [14] takes the network topology as an input and generates configuration files for BGP++.

- Network Simulation - Network Simulator (NS2) [15] is used for simulating the BGP network. NS2 requires a script which defines all the components in the network along with the connectivity information. It uses BGP++ as an agent for BGP simulations.

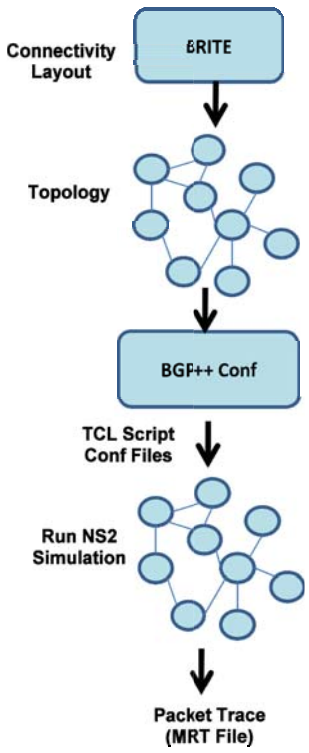


Figure 3: Generating BGP Traces

B. Multicore Simulation Environment

Once BGP traces are generated for a single node on the network, this node could be run on a multicore machine in order to analyze its performance. A *BGP Simulation Client* program is developed which reads BGP packets from the trace file and feeds it to the simulated node.

1) BGP Simulation Client

A BGP simulation client is a C program which imitates the functionality of several BGP peers and maintains connectivity with the actual BGP daemon. The client interacts with the TCP/IP stack of the host machine and initiates a BGP session (one for each peer) with the BGP Daemon. Once the

TCP session is established, it reads the packet dump file (the MRT format which was generated by performing BGP network simulation) and parses information from the file on a per peer basis. It then generates BGP packets based on the parsed data and sends them to the BGP Daemon. All the response packets from BGP Daemon are discarded. Once all the packets from the file are parsed and sent, the client terminates the TCP session with BGPD.

2) BGP Daemon

BGP Daemon is the actual node which is run on a simulated Multicore machine using Simics. BGP Daemon runs a parallel version of the Quagga routing software which was described in the previous section. Simics provides flexibility to vary the number of cores on the simulated machine. GEMS/Ruby memory model is also attached to the machine running BGP Daemon for simulating the memory hierarchy.

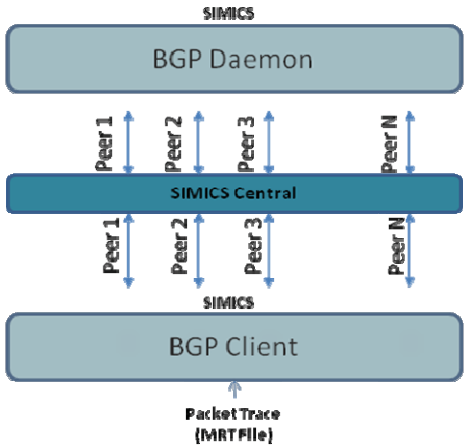


Figure 4: BGP Simulation Client

V. RESULTS AND DISCUSSION

A. Software Optimizations

This section presents the results and analysis of simulations performed on the Dynamic Task Scheduling scheme. We first compare the results of the baseline sequential implementation with the naïve DTS scheme and subsequently discuss how different optimizations help in achieving better performance speedups. We also evaluate the underlying multicore architecture and study the impact of various architecture parameters on the performance of the multithreaded application. The results are generated using a network topology involving 1000 nodes for processing 1000 updates messages. The update messages are sent to the node in consideration as fast as possible so as to evaluate the maximum update processing capability. Increasing the number of nodes or update messages translates to more number of network prefixes and hence larger routing tables. This effect has been studied in subsection V.B.5. and we conclude that processing time for 1000 updates is a representative number for comparing performance numbers.

1) Performance of naïve DTS scheme

First, we evaluate the performance of the naïve task scheduling scheme. The naïve implementation has only one set of read, write, ready and event queues on which all threads operate. Each peer has its own stream buffer which means only one packet per peer can be read at a time. A common set of file descriptors are used for probing TCP buffers using the select command. The performance of sequential code is shown in the first set of bars in figure 8. As expected, the execution time for the sequential code remains the same on varying the number of processor cores.

The second set of bars in figure 8 show the improvement in the execution time achieved with this scheme. Only the results for execution time on 16 processors are shown on the graph. On a lower number of processors, the lock contention among threads increases the execution time significantly and hence the numbers blow out of range. Analyzing the breakup of time consumed reveals that lock contention is a major performance bottleneck. Different optimization schemes are applied to reduce lock contention at different levels.

2) Eliminating Common Select Logic

Select logic is used for probing the TCP buffer for new packets. In the sequential implementation, a common file descriptor set (FDSET) is used to keep track of new packets from all peers. Keeping a common FDSET serializes the code acting as a logical barrier before each thread starts processing the new incoming packets. This has been totally avoided by assigning an independent file descriptor for each peer and each thread maintains its own FDSET which it uses to probe the TCP buffer for any new packets. As seen in figure 5, with this optimization, each thread can probe the TCP buffer independently and start processing them before waiting for other threads to finish execution.

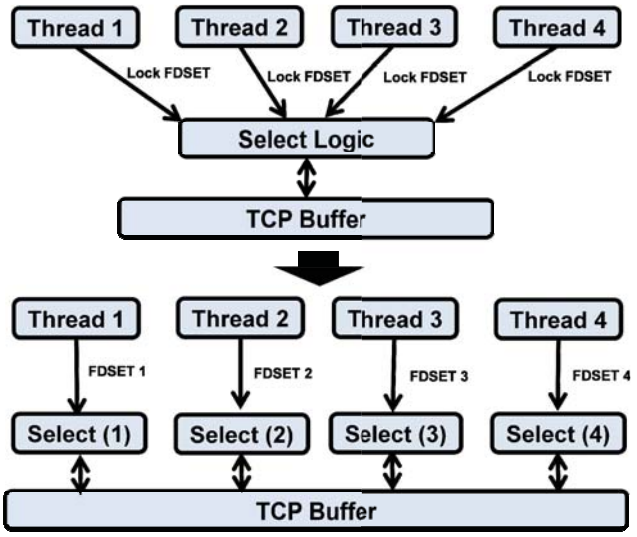


Figure 5: Eliminating common select logic

3) Thread Based Streams

We introduce the concept of thread based streams to resolve the lock contention on peer streams when multiple threads are operating on the same peer. When streams are associated with a peer, threads processing the same peer have to acquire lock to access these streams. If streams are maintained on a per thread basis, it allows each thread to access its stream independently of the others. There is an additional overhead requirement of storing these streams as the number of threads is greater than the number of peers.

For instance, if there are 10 peers and 40 threads, we would require an additional storage of 30 streams (30 * 512 bytes/stream ~ 15KB). However, the overhead of storing additional streams is marginal and can be ignored given the performance benefits that it provides. Figure 6 shows the removal of peer locks when streams are associated with threads instead of peers. As can be seen in the third set of bars in figure 8, thread based stream optimization gives significant speedup when run on different number of cores. The speedup achieved is approximately 2.4 times over the sequential implementation.

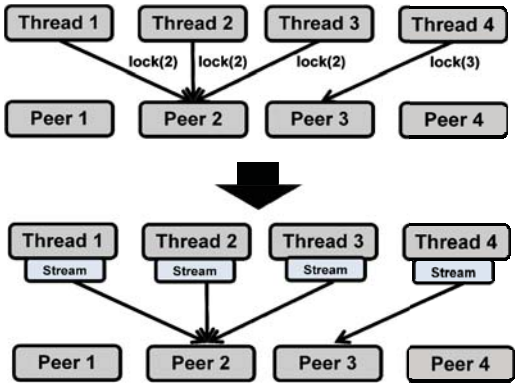


Figure 6: Thread Based Stream Optimization

4) Multiple Task Queues

In the naïve implementation of DTS scheme, only one set of task queues is maintained that is shared by all the threads in the BGP daemon which leads to contention on accessing these queues. As the queues are accessed frequently (every time a thread picks up a function to execute or schedules new functions to be executed), this becomes a severe performance bottleneck. An optimization to this bottleneck is to create a set of read, write, event and ready queues per thread instead of having a common set for all threads. Each thread picks up data from a different ready and event queue.

Data can be added to the multiple ready and event queues by several mechanisms. Two among such schemes were evaluated: one where tasks are allocated to queues in round robin fashion, and the other where an initial distribution is made across queues and all subsequent packets of the same peer are allocated to the same queue. In the round robin scheme, round robin variables are maintained for read, write and timer queues and tasks are added to the different queues by using the round robin variables as index. With this scheme, the round robin variables still have to be accessed in a critical section.

The later scheme with tasks being assigned on a peer basis eliminates the need for round robin variables as well and thus further reduces the lock contention. However, the cost of it is improper load balancing as some threads may have lot of tasks assigned to them while other threads may be idle. In our evaluation, the round robin scheme consistently performed better than the other one because of better load balancing at each thread. We thus chose the round robin scheme of assigning tasks for all out experiments. Figure 7 depicts the multiple task queues with round robin mechanism being used to assign tasks. Only the tasks being assigned to read queues are shown for illustration.

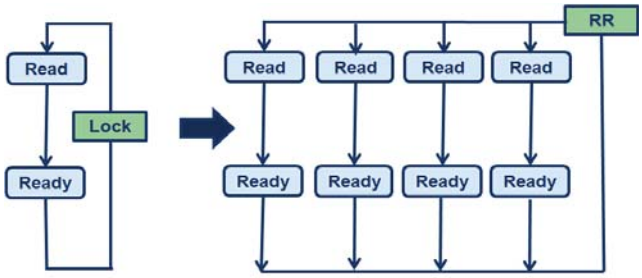


Figure 7: Multiple Task Queues with round robin scheme for adding tasks

The fourth set of bars in figure 8 show performance of the multithreaded implementation with multiple queues. As can be seen, this scheme significantly improves the performance over the last optimization. This gives a speedup of approximately 2.1 times over the last optimization and an overall speedup of 5.05 times over the sequential implementation for a multicore with 16 processors.

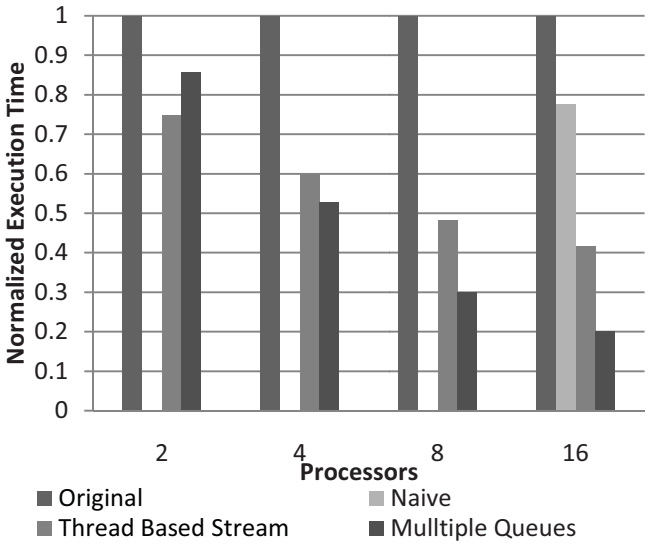


Figure 8: Execution Time with different optimizations on varying number of processor cores

B. *Architectural Analysis*

This section presents the architectural analysis done over the optimized multithreaded implementation described in Section 5.A. All the results are performed on a system with MOSI coherence protocol and a private L2 cache associated with each processor. The interconnection network has a hierarchical switch topology unless specified otherwise. Table 1 shows the generic configuration of the multicore architecture used for our simulations.

Multicore Parameters	
Processor	Ultrasparc III running Solaris 10
L1 Cache	Split I&D, 64 KB 4-way set associative, 1 cycle access latency, 64-byte line
L2 Cache	Private L2 per processor, 1 MB per processor, 6 cycle access latency, 64-byte line
On Chip Network	Hierarchical switch, 13 cycle link latency, 4 virtual channels, 4 buffers per virtual channel
Coherence	MOSI at L2 level
Ethernet	Link latency – 1000 CPU cycles

Table 1: Multicore Parameters

1) Varying Cache Sizes

We analyze the L1 and L2 cache behavior of the multithreaded implementation. Figure 9 shows the execution times for different L1 cache sizes. The application performance is very sensitive to the L1 cache size especially when the L1 cache is small. When the size is increased beyond 64 KB the performance saturates giving negligible speedup for subsequent doubling of the cache size.

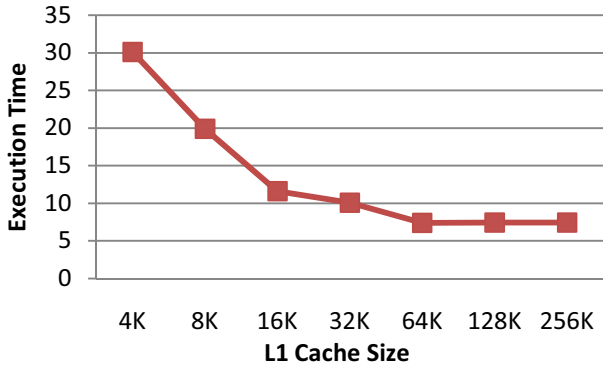


Figure 9: Execution Time (ruby cycles x 10^6) with varying L1 Size. (L2 cache is 1 MB.)

The application behavior on varying L1 cache size falls along the expected behavior. However, an interesting point to note is the L2 cache behavior. The performance of the DTS Scheme remains almost inelastic to L2 Cache size variation as shown in figure 10, where the cache size is increased from 64KB to 8MB per bank (1 bank per processor on a 16 processor system). The speedup achieved by adding a large on chip L2 cache is marginal when compared to the speedup seen by varying the L1 size. This behavior could be attributed to the fact that most data structures used in the implementation are allocated dynamically and hence do not exhibit any spatial locality. Moreover, all tables, lists and queues are organized as linked data structures and caches are known to perform poorly for such implementations.

As the application does not require large L2 cache size, we perform an optimization on L2 cache size by keeping it small and reducing the cache access latency to achieve performance improvement in memory accesses. This gives a further speedup of 2.9% over the optimized software implementation.

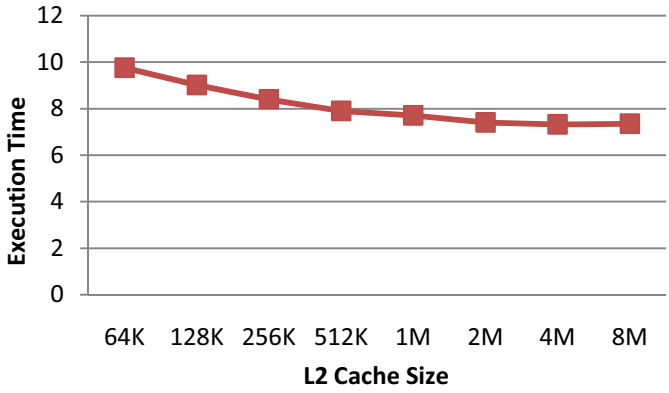


Figure 10: Execution Time (ruby cycles x 10^6) with varying L2 Size. (L1 cache is 64 KB.)

2) Private Vs Shared L2 Cache

We analyze the performance when the L2 cache is kept private to each core and when it is shared among all the processors in the system. The results are evaluated on a directory based memory system. The total L2 cache size of the system is kept constant. Figure 11 shows the performance comparison with both the multicore systems. As can be seen, the performance of a system with private L2 is better than that of a shared L2. For small number of cores, the performance difference is significant and the gap shrinks for large number of cores. The lower performance of shared L2 can be attributed to the fact that the application shows little data sharing (discussed in later sections) and hence keeping the L2 private eliminates the need for maintaining cache coherence at L2 levels thereby increasing the performance.

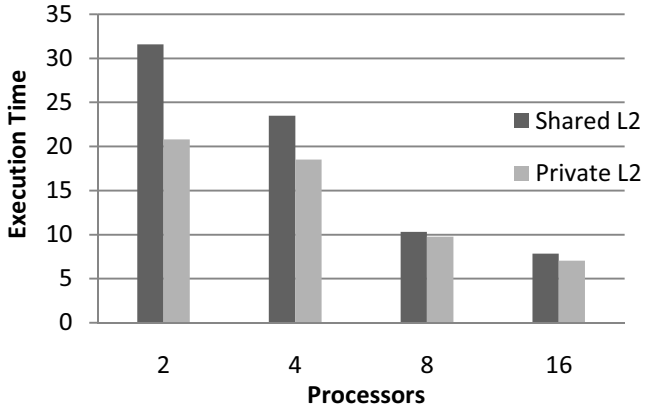


Figure 11: Execution Time (ruby cycles x 10^6) on Shared Vs Private L2 System

3) Directory Vs Broadcast based Memory System

Figure 12 shows the performance of the multithreaded application on a Directory and a Broadcast based memory system. A general behavior observed for this application is that there is not much data sharing between the processors. This could be attributed to the fact that all threads have dedicated data structures on which they operate. The only shared structure is the routing table which is large and each thread is usually updating different parts of the table. Hence, not many updates or invalidations happen during the simulation. In a broadcast based system, each cache sees all the memory requests, which is useful when there is lot of data sharing.

In our case, as most of the messages on the interconnect pertain to getting data from the memory, all caches don't need to see those messages, thereby limiting the amount of processing at the cache controllers. Hence, the directory based system performs better where each cache doesn't need to see all the traffic on the interconnection network and limits the number of messages processed by the cache controller.

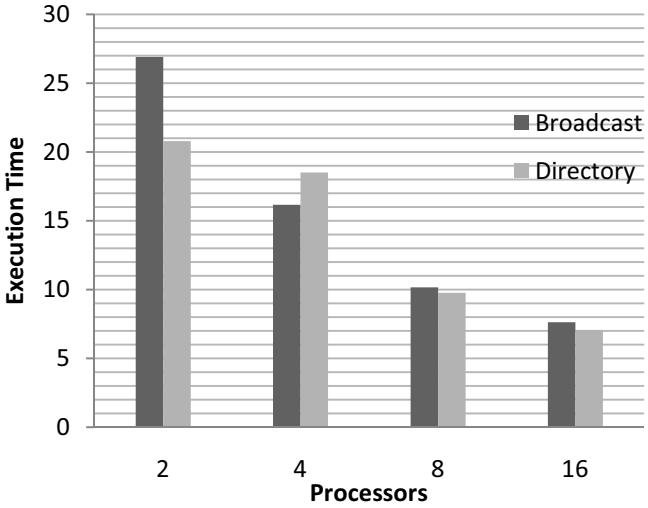


Figure 12: Execution Time (ruby cycles x 10^6) on Broadcast vs. Directory based system

4) Interconnection Network Behavior

We analyze the behavior of the multithreaded application on different interconnection network configurations. Figure 13 shows the execution time with varying link bandwidth on a system with 16 processors. The performance of the application is very sensitive when the link bandwidths are small but the performance gain saturates quickly. This shows that the application is not very sensitive to interconnection network bandwidth and links with low bandwidth are sufficient for achieving optimal performance.

Figure 14 shows the percentage link utilization for different interconnect bandwidths. The network utilization steadily falls as the network bandwidth is increased. The application does not require large interconnect bandwidth and complicated topologies to optimize the traffic and network consumption. Hence a simple low latency network could be used. This further reduces the execution time by 6.5 % on a 16 processor system giving an overall speedup of 9.35 % over the optimized software implementation.

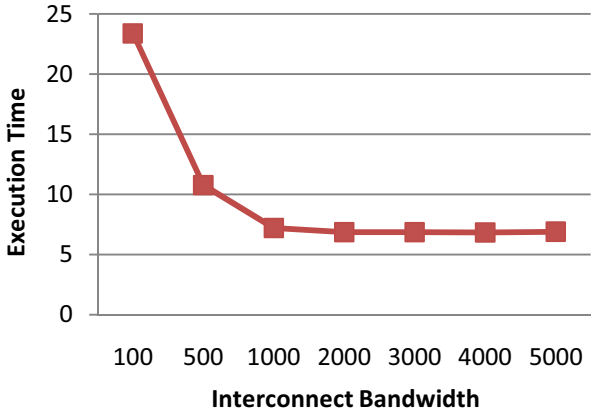


Figure 13: Execution Time (ruby cycles x 10^6) with varying link bandwidth (in bytes/cycle)

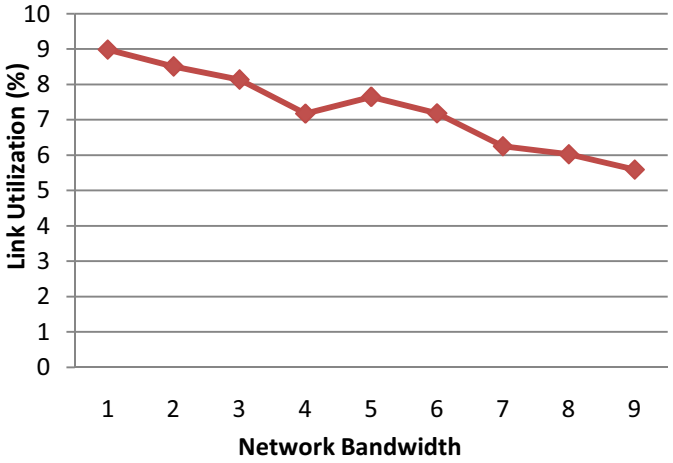


Figure 14: Percentage Link Utilization with varying link bandwidth (in Kbytes/cycle)

5) Large Network Simulation Traces

Simulations were run on larger trace sizes with each node advertizing four times the original number of IPV4 prefixes. This would lead to an increased routing table size and hence more time to traverse the routing tables. As the simulation progresses, the routing table sizes would grow and each update processing may take incrementally more time. However, the processing time remains almost constant as throughout the simulation. As can be seen in figure 15, the execution time for 500 updates remains approximately constant at about 3 million cycles.

The trend line in figure 15 shows the trend in update processing time as the simulation progresses. The line has a very small slope indicating that the processing times increase only slightly with larger routing tables. A reason for this may

be the organization of the routing table. Since it is a binary tree, having four times the number of prefixes would increase the depth of the tree by two or three levels, increasing the traversal time marginally as each level adds just one iteration to the tree traversal code. The increase in memory required to store the routing tables also does not have much impact on the processing time.

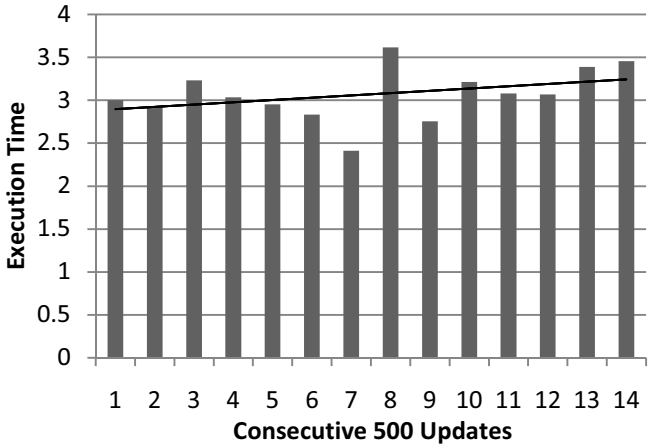


Figure 15: Execution Times (ruby cycles x 10^6) for 500 consecutive updates on 16 processors

6) Overall Performance Speedup

Finally, after all the above mentioned optimizations are applied, the best configuration is chosen and a speedup of 6.5x is achieved with 16 processors as seen in figure 16.

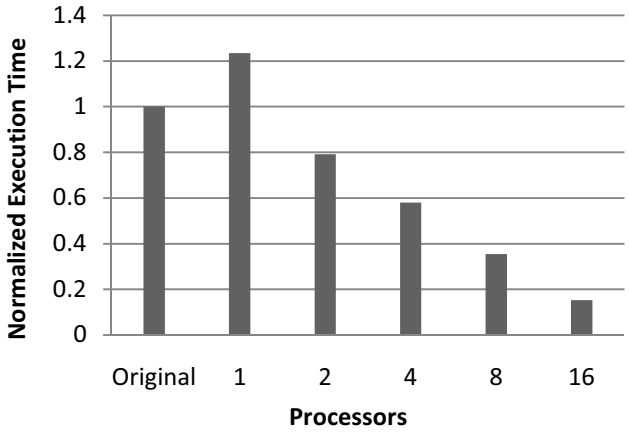


Figure 16: Overall Performance Speedup

7) DTS vs. PBTS

Figure 17 provides a comparison of speedups achieved with PBTS and DTS schemes. The numbers shown are normalized with respect to the execution time of the original sequential implementation. As can be seen, DTS performs better on different configuration because of better task allocation and load balancing among peers.

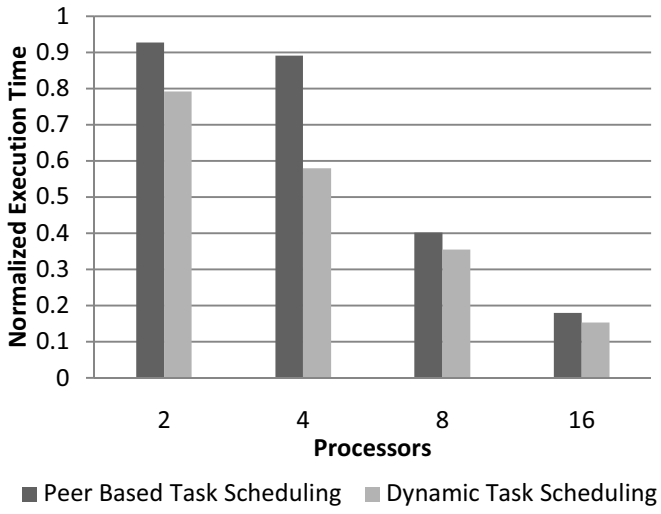


Figure 17: Comparison of Speedups achieved with PBTS and DTS schemes

VI. RELATED WORK

The use of distributed and multicore architectures in the data plane has been well researched. The control plane has caught attention only recently, with a variety of papers talking about different applications in the control plane, including network management, security, routing protocols etc. Most research in routing protocols has been done on speeding up BGP. Several techniques have been applied to extract parallelism in BGP on multiprocessor systems. Klockar et al [16] developed a distributed router based on modularized BGP. Zhang et al [17] propose another BGP model where route computations are done on multiple agents. There has also been an effort to run BGP on multicore using Thread Level Parallelism (TLP) [18]. More recent work in this domain has been focused on speeding up different aspects of the protocol but none of them provides a comprehensive mechanism to achieve overall performance improvement. Lei et al [19] develop a Threaded BGP implementation and combine it with a two-level trie based routing table structure for fast lookup in the routing tables.

While the works mentioned above achieve speedup for a variety of multicore systems and characterize certain aspects of application performance, none of them provides a comprehensive analysis of the major performance bottlenecks in the routing protocol implementations. Moreover, no prior work provides a perspective on programmability issues involved in writing multithreaded routing protocols. Our work fills that void by providing a comprehensive approach towards developing next generation routing protocols which could benefit from the increased processing power provided by multicore machines.

VII. CONCLUSION

The key contributions of this paper include providing comprehensive analysis of the generic software architecture of routing protocols and proposing ways on how it can be converted to a multithreaded application suitable for running on multicore processors. We also provide detailed architectural analysis of the multithreaded application and show several hardware level optimizations which can be used to improve application performance. We take Quagga BGP as a representative routing protocol implementation and achieve a speedup of 6.5 times speedup when run on 16 processor cores.

Although we do all our implementation and analysis on a particular implementation of a specific routing protocol, the parallelization techniques and analysis that we propose are applicable to other protocol implementations. For instance, Quagga routing suite also supports intra domain routing protocols namely OSPF and RIP. Implementations for these protocols use the same thread, stream and queue libraries that are used by BGP. In our implementation, we propose alternative implementations for stream and queue libraries. Making similar optimizations to OSPF and RIP implementations can give significant speedups in their performance. We view our work as a step towards the development of multithreaded routing protocols which could exploit the enormous computer power of multicore.

REFERENCES

[1] Internet Usage and Population Statistics, http://www.internetworldstats.com

[2] A Border Gateway Protocol 4, RFC 4271

[3] White Paper: “Packet Processing with Intel® Multi-Core Processors”, www.intel.com

[4] Cisco Quantumflow Processor, www.cisco.com

[5] BGP Report 2005, http://www.potaroo.net/papers/isoc/2006-06/bgpupds.html

[6] Quagga Routing Software Suite, GPL licensed IPv4/IPv6 routing software

[7] XORP Open Source Routing Platform, http://www.xorp.org/

[8] Milo M.K. Martin et. al., “Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset”, Computer Architecture News (CAN), September 2005.

[9] Virtutech Simics, www.simics.net

[10] The Cooperative Association for Internet Data Analysis, http://www.caida.org/home/

[11] Alberto Medina, et al. BRITE: An Approach to Universal Topology Generation. In Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- MASCOTS'01, Cincinnati, Ohio, August 2001.

[12] Xenofontas A. Dimitropoulos et al, Large-Scale Simulation Models of BGP, Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems

[13] http://tools.ietf.org/id/draft-ietf-grow-mrt-04.txt

[14] http://www.ece.gatech.edu/research/labs/MANIACS/BGP++/bgppp_conf.html

[15] Network Simulator 2, http://www.isi.edu/nsnam/ns/

[16] T. Klockar, M. Hidell, et al. Modularized bgp for decentralization in a distributed router, Winternet Grand Finale workshop, poster, 2005.

[17] X. Zhang, P. Zhu, X. Lu. “Fully-distributed and highly-parallelized implementation model of bgp4 based on clustered routers”. In: ICN 2005, LNCS 3421, Berlin: Springer-Verlag, pp.433-441, 2005

[18] Gao Lei et al, “Exploiting the Thread-Level Parallelism for BGP on Multi-core”, Communication Networks and Services Research Conference, 2008. CNSR 2008

[19] Gao Lei, Lai Mingche, Gong Zhenghu, "An Improved Parallel Access Technology on Routing Table for Threaded BGP," ICPADS, pp.198-204, 2009 15th International Conference on Parallel and Distributed Systems, 2009

Performance of Multi-Process and Multi-Thread Processing on Multi-core SMT Processors

Hiroshi Inoue and Toshio Nakatani

***Abstract*—Many modern high-performance processors support multiple hardware threads in the form of multiple cores and SMT (Simultaneous Multi-Threading). Hence achieving good performance scalability of programs with respect to the numbers of cores (core scalability) and SMT threads in one core (SMT scalability) is critical. To identify a way to achieve higher performance on the multi-core SMT processors, this paper compares the performance scalability with two parallelization models (using multiple processes and using multiple threads in one process) on two types of hardware parallelism (core scalability and SMT scalability). We tested standard Java benchmarks and a real-world server program written in PHP on two platforms, Sun's UltraSPARC T1 (Niagara) processor and Intel's Xeon (Nehalem) processor. We show that the multi-thread model achieves better SMT scalability compared to the multi-process model by reducing the number of cache misses and DTLB misses. However both models achieve roughly equal core scalability. We show that the multi-thread model generates up to 7.4 times more DTLB misses than the multi-process model when multiple cores are used. To take advantage of the both models, we implemented a memory allocator for a PHP runtime to reduce DTLB misses on multi-core SMT processors. The allocator is aware of the core that is running each software thread and allocates memory blocks from same memory page for each processor core. When using all of the hardware threads on a Niagara, the core-aware allocator reduces the DTLB misses by 46.7% compared to the default allocator, and it improves the performance by 3.0%.**

I. Introduction

Modern high-performance processors provide thread-level parallelism by using multiple hardware threads within each core using Simultaneous Multi-Threading (SMT)[1] [1]. For example, Sun's UltraSPARC T2 (Niagara 2) processor [2] provides 8-way multi-threading in each core, UltraSPARC T1 (Niagara) processor [3] and IBM's POWER7 processor [4] provide 4-way multi-threading, and Intel's Core i7 and Xeon (Nehalem) processors [5] provide 2-way multi-threading. Such SMT processors can increase the efficiency of CPU resource usage by allowing multiple threads to run on a single core, even for workloads with limited instruction-level parallelism. Multiple threads using SMT can also hide memory access latencies. Due to a growing gap between the processor and memory speed, SMT is becoming more important.

To exploit thread-level parallelism available in such a system, programs have to be parallelized. A programmer can write a parallel program using multiple processes (the *multi-process model*) or multiple threads within one process (the *multi-thread model*) on today's systems. Multiple threads in one process share the virtual memory space, while multiple processes do not share the same memory space. This difference typically results in a larger memory footprint and better inter-process isolation for the multi-process model. For example, the Apache Web server supports both a multi-process model (prefork model) and a multi-thread model (worker model). The prefork model generates multiple Web-server processes to handle many HTTP connections while the worker model generates multiple threads. The prefork model is more reliable but consumes more memory.

In this paper, we study the interactions between the parallelization model and the processor architecture aspects of a multi-core SMT processor. To identify how to achieve higher performance on the multi-core SMT processor, this paper focuses on performance comparisons between a multi-thread model and a multi-process model on two types of hardware parallelism (core scalability and SMT scalability). We measure the throughput of standard Java benchmarks, SPECjbb2005 and SPECjvm2008, using the multi-process model and the multi-thread model, and then we analyze the detailed architecture-level statistics such as cache misses and TLB misses on two processors, Sun's Niagara and Intel's Nehalem. The Niagara has eight cores and four SMT threads in each core, whereas the Nehalem has four cores and two SMT threads in each core. In the measured benchmarks, each thread did not communicate with each other so that we can focus on the architecture-level behavior.

To test the core scalability, we increased the number of cores with only one SMT thread in each core. To test the SMT scalability, we increased the number of SMT threads in each core. Although many previous studies [6-11] measured core scalability and SMT scalability of programs on multi-core SMT processors, this is the first work to focus on the effects of the parallelization model on multi-core SMT processors.

Our results showed that the multi-thread model achieved much better SMT scalability than the multi-process model because the multi-thread model generated a smaller number of cache misses and DTLB misses due to a smaller memory footprint. In contrast to better SMT scalability of the multi-thread model, on average, both models achieved almost

[1] A Niagara core cannot execute instructions from multiple software threads in the same processor cycle and thus the multi-threading in Niagara should be called *fine-grained multi-threading*. In this paper, we use the word SMT in a wider sense to include fine-grained multi-threading.

H. Inoue and T. Nakatani are with IBM Research – Tokyo, Kanagawa-ken 242-0001 Japan (e-mail: {inouehrs, nakatani}@jp.ibm.com).

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

comparable core scalability on Niagara and the multi-process model achieved better core scalability than the multi-thread model on Nehalem. We found that, contrary to our expectations, the multi-thread model generated up to 7.4 times more DTLB misses than multi-process model when we used many cores.

We confirmed that our observation was not specific to Java workloads by testing a real-world server program written in PHP on Niagara. The multi-thread model achieved better SMT scalability than the multi-process model by reducing the DTLB misses, but it showed comparable core scalability.

We also observed that the performance advantage of the multi-thread model was reduced as the number of cores increased. To achieve the better performance on the SMT processors using many cores with SMT threads, we implemented a new memory allocator in the multi-threaded PHP runtime. This allocator avoided sharing a page among threads running on different cores. This reduced the DTLB misses by about half and improved the performance when using all of the hardware threads on Niagara. The modern memory allocators are often aware of the remote and local memories on NUMA systems and control the allocations to minimize the costly remote memory accesses. Our allocator provided similar control at a finer granularity even on a non-NUMA system and improved the performance.

There are two main contributions in this paper. (1) We demonstrate that the smaller footprint of the multi-thread model does not always result in the higher performance and it may degrade the performance compared to the multi-process model due to more frequent DTLB misses on today's multi-core processors. The performance of the multi-thread model and the multi-process model depend on the underlying processor architecture and that the interactions are complicated. (2) We show that a memory allocator that tracks the processor core executing each program can improve the performance on a multi-core SMT processor by significantly reducing the DTLB misses.

The rest of the paper is organized as follows. Section 2 covers related work. Section 3 summarizes the experimental results with Java benchmarks. Section 4 describes our experiments with a larger PHP program. Section 5 describes our core-aware memory allocator and presents performance results. Finally, Section 6 summarizes this paper.

II. Related Work

There are many papers that have measured performance scalability on systems with high thread-level parallelism using multi-core SMT processors [6-11]. For example, Kaseridis and John [7] studied the scalability of the SPECjbb2005 on a Niagara processor. Later, Tseng *et al.* [8] also studied the scalability of Java programs on Niagara. Though these papers focused on the core scalability and the SMT scalability of the benchmarks, they did not study the differences of the multi-process model and the multi-thread model, which is our focus in this paper. Ueda *et al.* [9] studied the performance scalability of the SAP Java application server on a Niagara processor and on Intel's Xeon (Clovertown) processors, which do not support SMT, while changing the number of JVMs from one to four. The one JVM configuration in their work corresponds to the multi-thread model, and using more JVMs makes the program more like the multi-process model, which is an extreme case of the multiple-JVM configurations. They showed that using more JVMs eased the lock contention while it increased the memory footprint and cache misses. As a result, a 2-JVM configuration achieved the best performance on the Niagara system and a 4-JVM configuration peaked on the Xeon system. They concluded that the amount of cache memory was the key factor to determine the best configuration. Our results showed that the 4-way SMT of Niagara might be another important factor that reduced the performance of 4-JVM configuration on Niagara.

Many previous studies identified lock contention as the dominant cause of poor scalability on multi-core SMT processors [9-11]. For example, Ishizaki *et al.* [11] reported that the performance of Java server workloads running on a Niagara2 processor was improved by up to 132% by removing the contended locks. We found that appropriate use of the parallelization model is also an important factor in improving the performance of programs on multi-core SMT processors. The programs we used in this paper did not actually suffer from lock contention even when we used all of the hardware threads of a Niagara processor. This was because we selected programs that did not share data between the software threads so as to focus on interactions between parallelization model and underlying micro architecture. Server-side programs for Web applications, in which each thread serves a different client request without communicating among each other, are important examples of such programs that did not share data among software threads. Other important examples include Hadoop (an open-source implementation of the MapReduce runtime written in Java) or scientific workloads using MPI.

In this paper, we propose a new memory allocator for multi-threaded programs, an allocator that is aware of the physical core that is running the software threads and allocates memory from the same memory page for each core. The basic idea of such a memory allocator is not new. For example, existing NUMA-aware memory allocators [12] are aware of the physical locations of the software threads and allocate memory blocks from local memory on the NUMA machines. We use a similar technique for a different target.

III. Performance Results for Java Workloads

A. System Setup

To study the performance of programs on a multi-core SMT processor, we choose Sun's UltraSPARC T1 (Niagara) processor, which has 8 cores with 4 SMT threads in each core, and Intel's Xeon X5570 (Nehalem) processor, which has 4

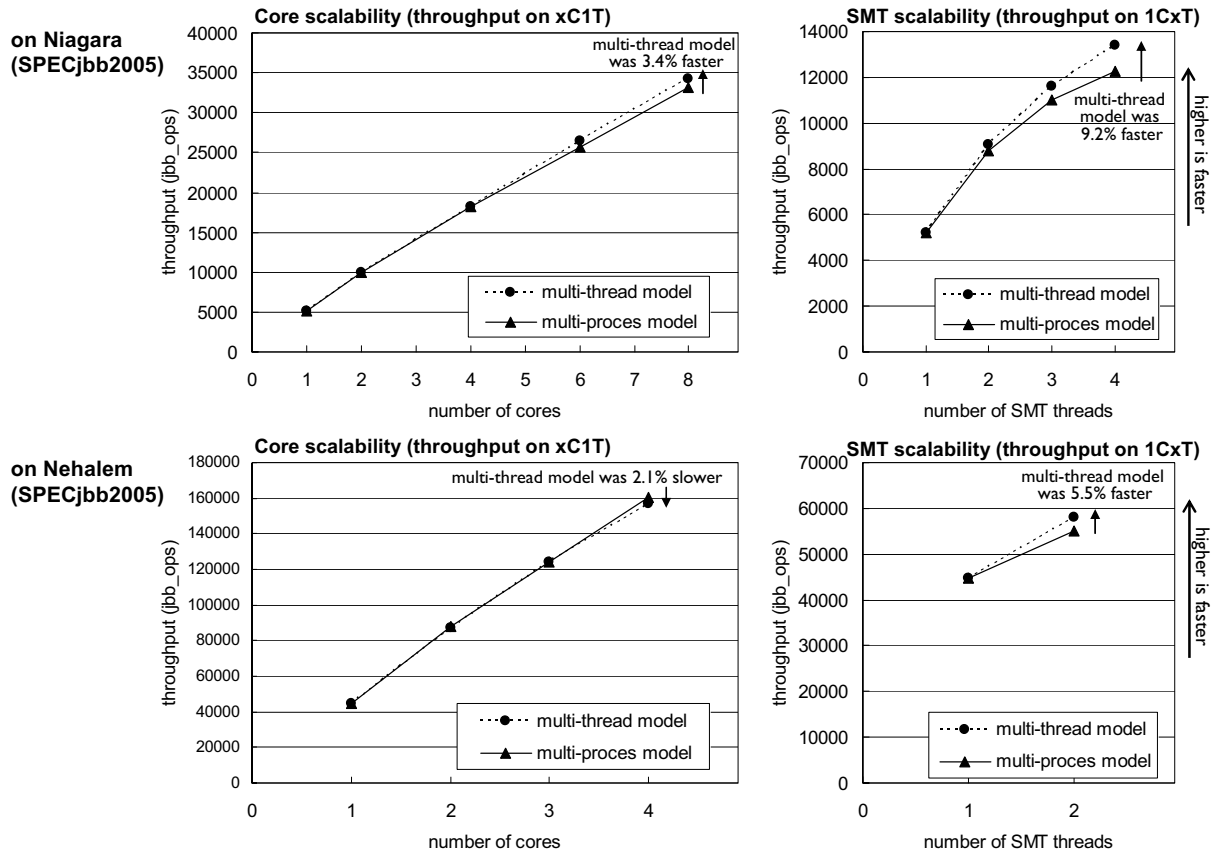


Figure 1. Core scalability and SMT scalability for SPECjbb2005 on Niagara and Nehalem.

cores with 2 SMT threads in each core, as the platform for our evaluations. A Niagara core contains 16 KB of L1 instruction cache, 8 KB of L1 data cache, and 64 entries in its DTLB. The 8 cores all share a 3-MB unified L2 cache. We used a system with a 1.2-GHz Niagara processor and 16 GB of system memory, running Solaris 10. A Nehalem core contains 32 KB of L1 instruction cache, 32 KB of L1 data cache, 256 KB L2 cache, 64 entries for small pages and 32 entries for large pages in a L1 DTLB, and 512 entries for small pages in a unified L2 TLB. The all cores share an 8-MB L3 cache. We used a system with a 2.93-GHz Nehalem processor and 24 GB of system memory, running RedHat Enterprise Linux 5.4.

In this section, we used two standard Java benchmarks, SPECjbb2005 [13] and selected programs from SPECjvm2008 [14]. We picked these benchmarks because they do not share data among the software threads and thus we can run them with the multi-process model and the multi-thread model without having to modify the programs. For example, in SPECjbb2005, which emulates a server side Java workload in a three-tier system, a worker thread accesses only its own Warehouse data set and does not communicate with other threads. Though the default configuration of the SPECjbb2005 is a multi-thread model using one JVM, it also allows the use of multiple JVMs. Thus we can run the SPECjbb2005 as a multi-process model by setting the same value as the number of hardware threads for the number of JVMs. In SPECjvm2008, worker threads do not communicate with each other and hence we can execute these programs in the multi-process model, though such multiple-JVM configuration is not officially allowed to submit the results. When a program requires temporary disk space, we explicitly specify a different location for each JVM to avoid conflicts among JVMs. We used the 32-bit HotSpot server VM for Java 6 Update 17 on both platforms. We configured the size of the Java heap as 256 MB per software thread. We specified a memory page size of 4 MB (Niagara) or 2 MB (Nehalem) for the Java heap in the JVM command-line options. To control the number of cores and SMT threads used, we explicitly specify the logical processors to use for each JVM by using the tools provided by the operating systems, psrset on Solaris and numactl on Linux.

B. Core Scalability and SMT scalability

In this section, we describe our experimental results for the core scalability and SMT scalability of the benchmarks with the multi-thread model and the multi-process model. In these results, we use the notation xCyT to show that x cores with y hardware threads per core were used in the experiment [8]. For example, 8C4T means that all hardware threads of a Niagara processor were used in the measurement and 1C1T means only one hardware thread on one core was used.

Figure 1 shows the core scalability and SMT scalability of SPECjbb2005 on two platforms. For the SMT scalability on

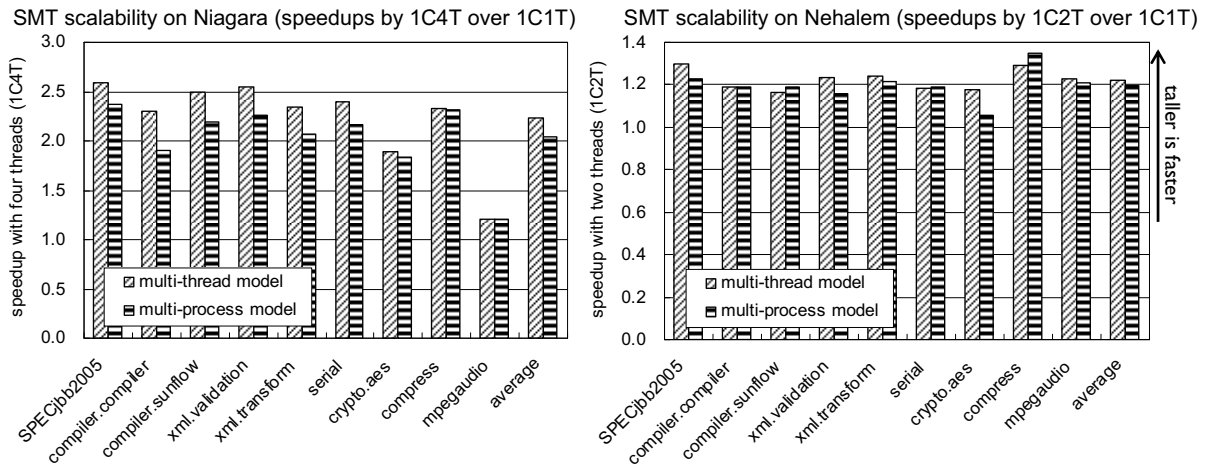


Figure 2. SMT scalability using multiple SMT threads on one core on Niagara (1C4T) and Nehalem (1C2T) for SPECjbb2005 and SPECjvm2008.

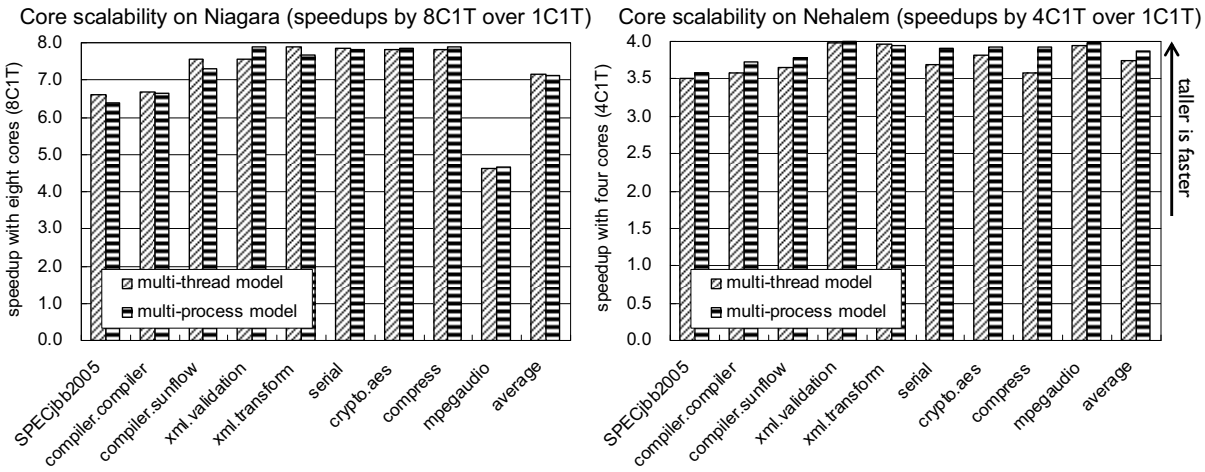


Figure 3. Core scalability using one thread/core on Niagara (8C1T) and Nehalem (4C1T) for SPECjbb2005 and SPECjvm2008.

Niagara, the multi-thread model outperformed the multi-process model when using two or more SMT threads. The maximum performance advantage of the multi-thread model was 9.2% when using four SMT threads (1C4T). For the core scalability on Niagara, the multi-thread model and multi-process model achieved almost comparable speedups for up to four cores. When all eight cores were used, the multi-thread model was 3.4% faster than the multi-process model. We detail the reason in the next section using architecture-level statistics. On Nehalem, the performance difference of the multi-thread model and the multi-process model was smaller. The multi-thread model was faster than the multi-process model using two SMT threads (1C2T) by 5.5% while it was slower using all four cores with one thread each (4C1T) by 2.1%.

To compare the SMT scalability and the core scalability with the multi-thread model and the multi-process model for all workloads, Figure 2 depicts the speedups using all four SMT threads (1C4T) on a Niagara core or all two SMT threads on a Nehalem core (1C2T). On both platforms, the multi-thread model achieved larger speedups using SMT threads (better SMT scalability) than the multi-process model. The performance advantages were 9.6% on Niagara and 2.0% on Nehalem on average of the tested programs.

Figure 3 shows the speedups using eight Niagara cores (8C1T) or four Nehalem cores (4C1T) with one thread on each core. Unlike the SMT scalability, the core scalability with the multi-thread model was not better than the core scalability with the multi-process model on both platforms. The performance advantage of the multi-thread model was almost negligible on Niagara and the multi-thread model was 3.0% slower than the multi-process model when using all cores of each processor. In Figure 2 and 3, poor scalabilities in mpegaudio on Niagara were due to the contentions in a floating-point unit because Niagara equipped only one floating-point unit shared by all cores.

C. Architecture-Level Statistics

For further insight into the causes of the differences in the core scalability and the SMT scalability, Figure 4 shows the change in the relative numbers of instructions and cache misses per transaction for the multi-thread model over the multi-process model for SPECjbb2005 on Niagara and Nehalem as we changed he numbers of cores and SMT threads. The bars that are shorter than 1.0 mean that the

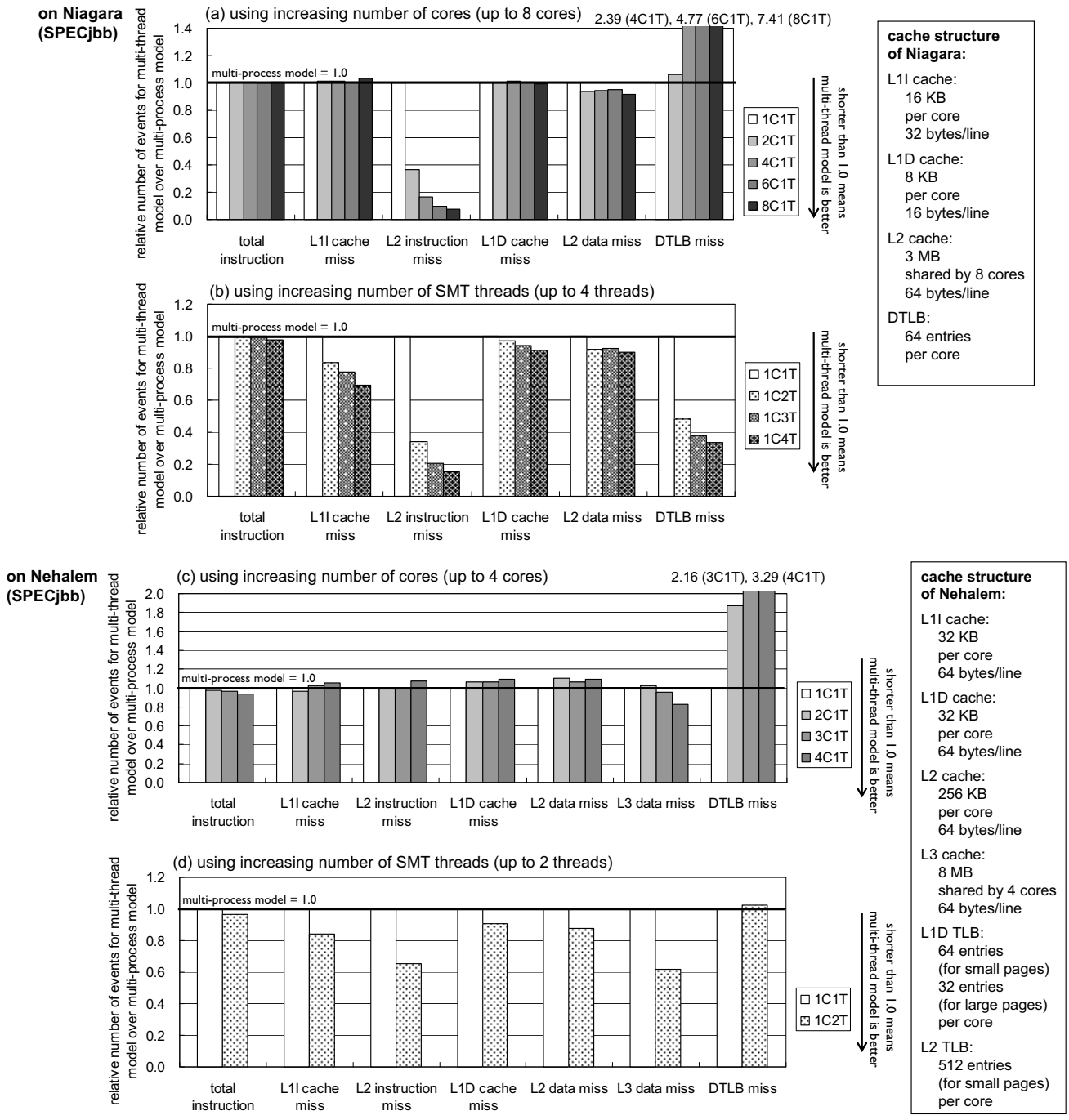


Figure 4. Comparisons of the numbers of instructions and cache misses with multi-thread model against multi-process model for SPECjbb2005 with (a) increasing numbers of cores on Niagara, (b) increasing numbers of SMT threads on Niagara, (c) increasing numbers of cores on Nehalem and (d) increasing numbers of SMT threads on Nehalem.

multi-thread model generated fewer cache misses than the multi-process model and hence the multi-thread model was better.

In Figure 4(a), the numbers of instructions and L1 cache misses were not affected by the parallelization model, whereas the numbers of L2 cache data misses and L2 cache instruction misses were smaller for the multi-thread model. These reduced L2 cache misses were due to the smaller footprint of the multi-thread model. In particular, the reductions in the L2 cache instruction misses were significant. This was because the JVM had a JIT compiler, and thus the instructions generated by the JIT compiler at runtime were not shared among the processes and the instruction footprint increased in proportion to the number of JVMs used.

For the DTLB misses, contrary to our expectations, the multi-thread model generated up to 7.4 times more misses than the multi-process model, though the total memory footprint was smaller for the multi-thread model. This notable increase in DTLB misses was because each worker thread and GC thread needed to access the entire memory space of a

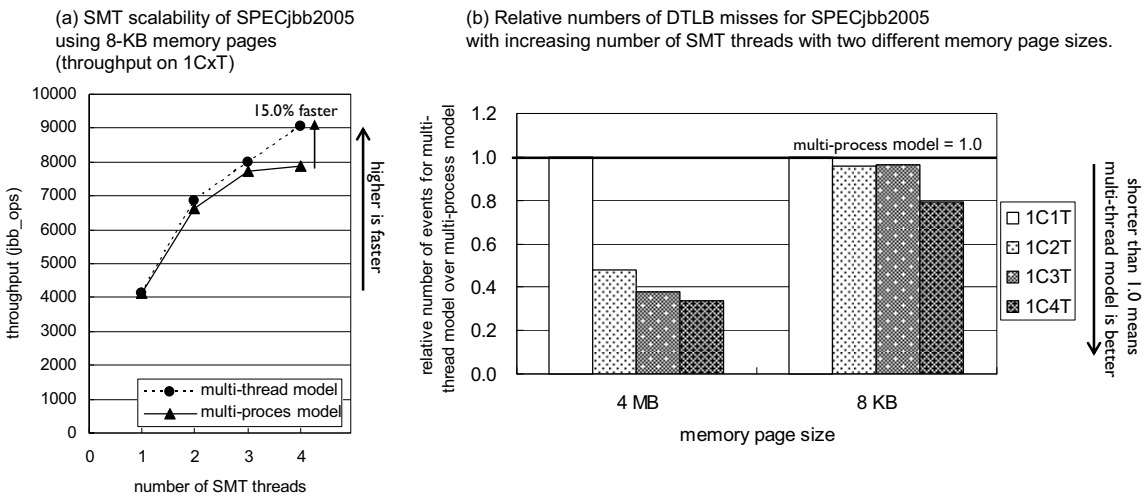


Figure 5. (a) SMT scalability using 8-KB memory pages for SPECjbb2005 on Niagara and (b) relative numbers of DTLB misses with increasing number of SMT threads with two different memory page sizes.

large Java heap (256 MB x the number of threads). Such access patterns resulted in more DTLB misses, because each core needed to preserve the DTLB entries for the entire memory space. In contrast, in the multi-process model each process accessed only its own 256-MB Java heap. A core executing a process needed to preserve only the DTLB entries for a limited memory area and thus the number of DTLB misses was not increased when using more cores. The overall effect of the reduced L2 misses and the increased DTLB misses was that the multi-thread model achieved slightly better core scalability than the multi-process model for SPECjbb2005. From Figure 3, the multi-process model outperformed the multi-thread model in some benchmarks. In these benchmarks, the effect of increased DTLB misses was larger than the reduced cache misses.

On the contrary, as shown in Figure 4(b), the number of DTLB misses for the multi-thread model became smaller than for the multi-process model as we increased the number of SMT threads. Because the SMT threads on one core shared the DTLB, the number of DTLB misses was not increased even when each thread accessed the entire Java heap. The number of L1 and L2 cache misses was smaller for the multi-thread model than for the multi-process model due to the smaller memory footprint of the multi-thread model. As a result, the multi-thread model achieved much better SMT scalability than the multi-process model as shown in Figure 1.

On Nehalem, as shown in Figure 4(c), the number of DTLB misses for the multi-thread model became significantly larger than for the multi-process model when we increased the number of cores. However, the multi-thread model and the multi-process model generated almost comparable number of DTLB misses as shown in Figure 4(d) when we used both SMT threads of a Nehalem core. The multi-thread model reduced L1, L2 and L3 cache misses and these reduced cache misses resulted in a higher SMT scalability of the multi-thread model. Note that L1 and L2 cache are private for each core and L3 cache is shared among cores on Nehalem, while L2 cache is shared on Niagara.

To confirm that our observations do not specific for HotSpot JVM, we conducted the same evaluations using 64-bit IBM JVM for Java 6, which uses mark-and-sweep GC with flat (non-generational) Java heap while HotSpot VM uses generational GC, on Nehalem. The results with IBM JVM were consistent with the results with HotSpot. For example in SPECjbb2005, the multi-thread model achieved 3.3% better performance than multi-process model using two threads (1C2T), while it was 3.4% slower using four cores (4C1T). We also observed the multi-thread model generated 1.94x more DTLB misses compared to the multi-process model using four cores (4C1T).

To examine how the memory page size affects our observations, Figure 5(a) shows the SMT scalability of SPECjbb2005 on Niagara using 8-KB memory pages instead of 4-MB pages. The multi-thread model achieved higher SMT scalability than the multi-process model even with the smaller pages. The performance advantage of the multi-thread model was increased from 9.2% (with 4-MB pages) to 15.0% (with 8-KB pages).

Figure 5(b) shows the relative number of DTLB misses of the multi-thread model over the multi-process model as we increased the number of SMT threads with the two memory page sizes. We did not include the counts of L1 and L2 cache misses in the figure because they were not significantly affected by the page size. The difference in DTLB misses with the multi-thread model and the multi-process model was smaller for the 8-KB pages than for the 4-MB pages. The multi-thread model generated about 20% fewer DTLB misses than the multi-process model even with 8-KB memory pages. Though this looks smaller than the difference when we used 4-MB pages, the absolute number of total DTLB misses was drastically increased with the smaller page size and hence the difference of 20% had a large impact on performance. For example, the number of DTLB misses per instruction was 0.018% for the multi-thread model with 4-MB pages and

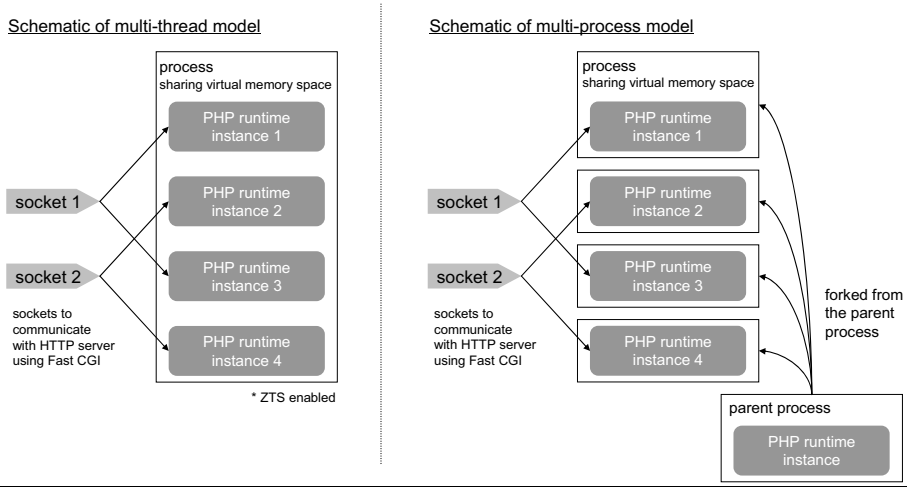


Figure 6. Structures of the multi-thread model and multi-process model for the PHP runtime.

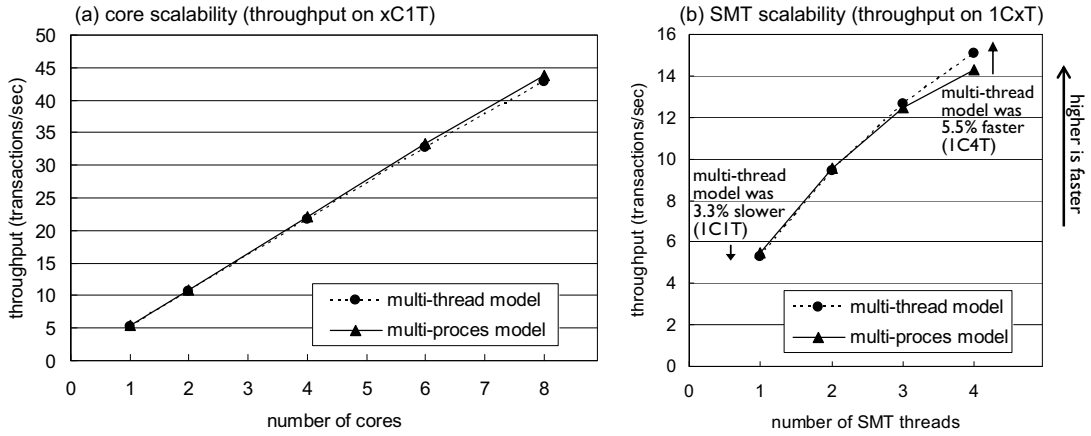


Figure 7. (a) Core scalability using one thread/core and (b) SMT scalability using multiple SMT threads on one core for MediaWiki on Niagara.

0.86% for the multi-thread model with 8-KB pages when using 4threads on 1 core (4C1T).

Though we do not show these results, the multi-process model achieved up to 4.2% better core scalability due to the smaller number of DTLB misses when we used 8-KB pages for the Java heap, while the multi-thread model showed up to 3.4% better core scalability when we used 4-MB pages.

IV. Performance Results for a PHP Workload

A. System Setup

To confirm that our observations shown in Section III are not specific to Java workloads, we tested a real-world server workload, MediaWiki [15], written in PHP. MediaWiki is a wiki server program developed by the WikiMedia foundation for the Wikipedia online encyclopedia. We imported 1,000 articles from a Wikipedia database dump (http://download.wikimedia.org/). We configured MediaWiki to use memcached-1.2.1 running on the same machine to cache the results of the database queries. We measured the throughput while reading randomly selected articles. We used PHP-5.2.1 [16] for the runtime, lighttpd-1.4.13 for the HTTP server, and mysql-4.1.20 for the database server. We also installed the APC-3.0.14 extension to the PHP runtime, which enhances the performance of PHP applications by caching the compiled intermediate code. The database server and the client emulator ran on separate machines. The number of PHP runtime instances (process or thread) was calculated from the number of hardware threads used for the evaluation by multiplying by 1.5 to hide the latency to access the backend database and fully utilize the processor. Hence we used 48 processes or threads when we used all 32 hardware threads in the processor. The emulated clients paused for three seconds as 'thinking time' between each request. The number of emulated clients was set for the highest throughput that allowed the average response time to stay under 2.0 seconds.

Though the default configuration of the original PHP runtime uses the multi-process model, it also supports the multi-thread model by enabling a compile-time option called ZTS. Figure 6 shows the structure of the multi-thread model and the multi-process model for the PHP runtime. Each PHP runtime instance handles independent requests and there is no communications among the instances. The original ZTS implementation used the pthread library for thread-local memory. To improve the performance, we modified the PHP runtime to use the thread-local construct (__thread) provided by the Sun C compiler instead of the pthread library, because

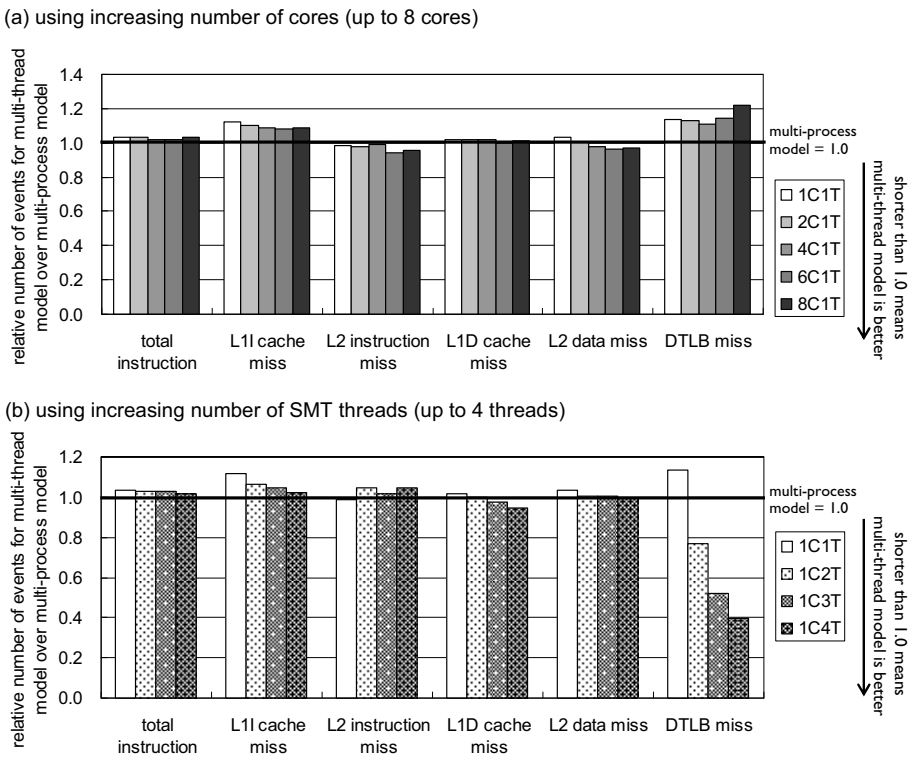


Figure 8. Comparisons of the numbers of instructions and cache misses with multi-thread model against multi-process model for MediaWiki (PHP) on Niagara while (a) increasing the number of cores and (b) increasing the number of SMT threads.

the pthread library has larger overhead for better portability. We also modified the PHP runtime to use multiple UNIX domain sockets to communicate with the HTTP server. We did not need to change the MediaWiki source code to run the program in the multi-thread model. We explicitly specified a memory page size of 4 MB for the heap area by using the MPSS (Multiple Page Size Support) library.

B. Core Scalability and SMT scalability

Figure 7 shows the core scalability and SMT scalability of MediaWiki. For the core scalability, both models achieved almost linear speedup with the increase in the number of cores used, although the multi-process model outperformed the multi-thread model from 2.0% to 3.3% regardless of the number of cores. The PHP runtime was originally optimized for a multi-process model and thus enabling the multi-threading support adds overhead for extra indirections to access global variables. The multi-process model was faster when using only one SMT thread per core due to this additional overhead. For the SMT scalability, however, the multi-thread model showed better scalability with increasing numbers of SMT threads and outperformed the multi-process model when using three or four SMT threads. The multi-thread model was slower by 3.3% when using only one SMT thread (1C1T) due to the thread safety overhead but it became faster than the multi-process model by 5.5% when using four SMT threads (1C4T) because of the benefit of reduced DTLB misses and cache misses in the multi-thread model. This advantage was smaller than the advantage of 9.5% for Java workloads shown in Figure 2. This was partially because the JVM inherently supported multiple threads and thus there was no additional overhead due to the multi-threading support.

Figure 8 shows the relative number of instructions and cache misses per transaction for the multi-thread model over the multi-process model in MediaWiki. In Figure 8(a), the number of instructions per transaction was slightly larger for the multi-thread model than for the multi-process model due to the additional instructions to make the runtime thread-safe. Supporting thread-safety also increased the number of cache misses. By increasing the number of cores, the gap between the multi-thread model and multi-process model in the number of L2 cache misses was slightly reduced due to the smaller memory footprint of the multi-thread model. When using four or more cores, the multi-thread model generated fewer L2 cache misses in spite of the additional indirect accesses for thread safety. In contrast, only the gap in the number of DTLB misses was increased by increasing the number of cores. When multiple software threads running on separate cores shared the same memory page, each thread was able to use only a part of the page, so the number of the required memory pages was increased, causing an increase in the DTLB misses. As a result of the increased DTLB misses and fewer L1 and L2 cache misses with more cores, the multi-thread model achieved scalability almost comparable to the multi-process model. In Figure 8(b), the relative number of DTLB misses was drastically decreased with the use of more SMT threads. Due to this large reduction in DTLB

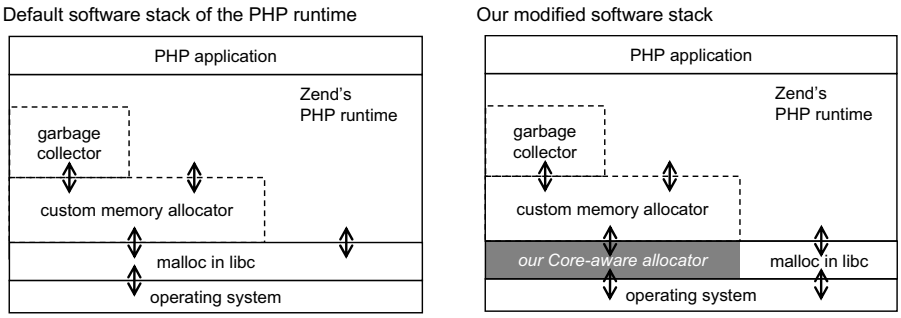


Figure 9. Software stacks of the PHP runtime with and without our core-aware allocator. Only the part shown in gray was modified to use the core-aware allocator.

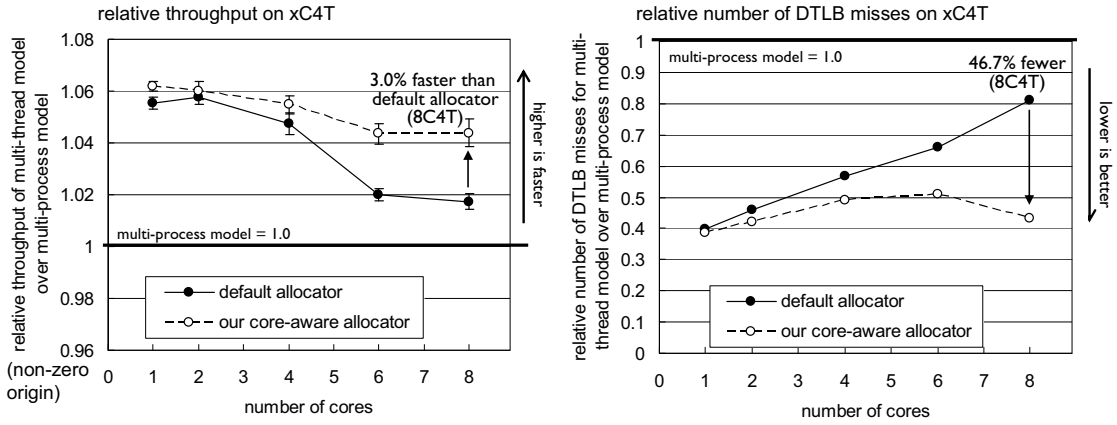


Figure 10. Relative throughput and the number of DTLB misses of the multi-thread model over the multi-process model with and without our core-aware allocator for MediaWiki on Niagara. Error bars show 95% confidence intervals.

misses, the multi-thread model achieved much better SMT scalability than the multi-process model, while its core scalability was roughly equal to the multi-process model.

V. Core-Aware Memory Allocation

A. Motivation

As described in the previous sections, the multi-thread model achieved higher performance on each core with multiple SMT threads. However we found that the performance advantage of the multi-thread model was reduced as the number of cores increased. For example, the performance advantage of the multi-thread model over the multi-process model for MediaWiki was only 1.7% when using all eight cores, whereas it was 5.5% on one core. The difference in the DTLB miss ratio was also decreased by increasing the number of cores. As shown in Figure 8(a), the multi-thread model tended to increase the number of DTLB misses by using more cores. One naive technique to avoid such increases in DTLB misses is to use multiple processes and bind each process to a specific core. However, this naive technique often suffers from imbalanced load among the cores and increased memory use.

B. Implementation

To maximize the performance of the multi-threaded programs on many SMT processor cores by further reducing the number of DTLB misses, we implemented a memory allocator in the multi-threaded PHP runtime. We call this allocator the core-aware allocator. The core-aware allocator tracks which core executed each requester (a software thread calling the allocator) and allocates memory blocks from the same memory page for each core. The idea of a memory allocator being aware of the physical location of the requester is not new. For example, most of the memory allocators used in modern UNIX systems support local memory allocation in NUMA systems to avoid costly remote memory accesses. Tikir and Hollingsworth [11] also implemented a NUMA-aware memory allocator in a JVM. With our core-aware allocator, we seek to show that a memory allocator that is aware of the physical location of the requester at a finer granularity can improve the performance of the running programs even on a non-NUMA system. Though our allocator is not specialized for SMT processors, the effect of the DTLB misses is typically larger on processors with many SMT threads and thus our allocator is most suitable for exploiting multi-core SMT processors.

The current PHP runtime uses a custom memory allocator. The custom memory allocator requests fixed-size (256 KB by default) memory chunks to the underlying memory allocator. The custom memory allocator requests a new chunk if a request cannot be handled in the already allocated memory chunks. The underlying memory allocator can use malloc in libc (the default) or an mmap system call in UNIX-based

systems. We implemented our allocator as an underlying memory allocator and the PHP runtime calls it instead of the malloc in libc. We did not modify the custom memory allocator of the PHP runtime itself. Figure 9 illustrates our modifications to the PHP runtime.

Our allocator obtains at startup time a 256-MB memory block for each core using an mmap system call. We explicitly specify 4 MB as the page size. Our allocator treats the 256-MB memory block as a pool of 1024 memory chunks of 256 KB each because each request from the custom allocator in the PHP runtime is always a multiple of 256 KB. When our allocator receives a request, it returns a memory chunk from the pool for the core which is running the requester. To identify the core with the requester, we use the lwpsinfo file for the requester thread in the proc file system. To avoid excess overhead in accessing this file system, we only access the lwpsinfo file once per Web transaction, when the PHP runtime receives a new transaction from the HTTP server. If an operating system frequently moves threads to different cores, we would need to increase the frequency of these location checks.

C. Performance with our core-aware allocator

Figure 10(a) shows how our core-aware allocator implemented in the multi-threaded PHP runtime improved the throughput. We conducted the measurement four times and averaged the results. The reduction in the performance of the multi-thread model was successfully avoided with our core-aware allocator. The multi-thread model with the core-aware allocator outperformed the multi-process model by 4.4% and the multi-thread model with the default allocator by 3.0% when we used all of the hardware threads in Niagara (8C4T). To examine the causes of the differences with and without the core-aware allocator, Figure 10(b) shows the relative number of DTLB misses over the multi-process model with and without the core-aware allocator. The core-aware allocator avoided increase in the number of DTLB misses with increases in the number of cores. The reduction in DTLB misses with the core-aware allocator when using eight cores (8C4T) was 46.7%. These results showed that a memory allocator should track which core is executing a program to maximize the performance of the running programs on multi-core SMT processors.

VI. Summary

In this paper, we studied the interactions between the parallelization model and the processor architecture aspects of a multi-core SMT processor. We evaluated the performance and the detailed architecture-level statistics of programs implemented in the multi-process model and the multi-thread model. Our results showed that both models achieved almost comparable core scalability, whereas the multi-thread model achieved much better SMT scalability and higher performance. We showed that the DTLB misses were the key to the differences between the core scalability and the SMT scalability.

Although the multi-thread model showed significant advantages on one core with multiple threads, the gain was reduced as the number of cores increased. To maximize the performance on the multi-core SMT processors, we implemented a memory allocator for a multi-threaded PHP runtime to reduce DTLB misses on multi-core SMT processors. Our allocator was aware of the core that is running the requester and allocates memory from the same memory page for each core. This reduced the DTLB misses by 46.7% compared to the multi-threaded PHP runtime without our allocator, and performance was improved by 3.0%. These results showed that a memory allocator should track which core is executing a program to maximize the performance of the running programs on SMT processors.

References

[1] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In Proceedings of the Annual International Symposium on Computer Architecture, pp. 392–403, 1995.

[2] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, A. Wynn. UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC. In Proceedings of the IEEE Solid-State Circuits Conference, pp. 22–25, 2007.

[3] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. IEEE Micro 25(2), pp. 21–29, March/April 2005.

[4] R. Kalla. POWER7: IBM's Next Generation POWER Microprocessor. A Symposium on High Performance Chips 21, 2009.

[5] R. Singhal. Inside Intel Core Microarchitecture (Nehalem). A Symposium on High Performance Chips 20, 2008.

[6] R. Iyer, M. Bhat, L. Zhao, R. Illikkal, S. Makineni, M. Jones, K. Shiv, D. Newell. Exploring Small-Scale and Large-Scale CMP Architectures for Commercial Java Servers. In Proceedings of IEEE International Symposium on Workload Characterization, pp. 191–200, 2006.

[7] D. Kaseridis and L. K. John. CMP/CMT Scaling of SPECjbb2005 on UltraSPARC T1. Workshop on Computer Architecture Evaluation using Commercial Workloads. 2007.

[8] J. H. Tseng, H. Yu, S. Nagar, N. Dubey, H. Franke, P. Pattnaik, H. Inoue, and T. Nakatani. Performance studies of commercial workloads on a multi-core system. In Proceedings of IEEE International Symposium on Workload Characterization, pp. 57–65, 2007.

[9] Y. Ueda, H. Komatsu, and T. Nakatani. Scalability Study of a Java Application Server on Two Multi-core Systems. Workshop on Computer Architecture Evaluation using Commercial Workloads, 2008.

[10] M. Ohara, P. Nagpurkar, Y. Ueda, and K. Ishizaki. The Data-centricity of Web 2.0 Workloads and its Impact on Server Performance. In Proceedings of the International Symposium on Performance Analysis of Systems and Software, pp. 133–142, 2009.

[11] K. Ishizaki, S. Daijavad, and T. Nakatani, Analyzing and Improving Performance Scalability of Commercial Server Workloads on a Chip Multiprocessor, In Proceedings of the IEEE International Symposium on Workload Characterization, pp. 217–226, 2009.

[12] M. M. Tikir and J. K. Hollingsworth. NUMA-Aware Java Heaps for Server Applications. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium, pp. 108, 2005.

[13] Standard Performance Evaluation Corporation. SPECjbb2005. http://www.spec.org/jbb2005/ .

[14] Standard Performance Evaluation Corporation. SPECjvm2008. http://www.spec.org/jvm2008/ .

[15] Wikimedia Foundation. MediaWiki. http://www.mediawiki.org .

[16] The PHP Group. PHP: Hypertext Preprocessor. http://www.php.net/ .

Author Index

NOTES