DESIGNING AN EFFECTIVE HYBRID TRANSACTIONAL MEMORY SYSTEM

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Chí Cao Minh
September 2008

© Copyright by Chí Cao Minh 2008
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Christos Kozyrakis) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Kunle Olukotun)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Subhasish Mitra)

Approved for the University Committee on Graduate Studies.

Abstract

Multi-core chips are now commonplace in server, desktop, and even embedded systems; however, they create an inflection point for mainstream software development. To benefit from the additional performance offered by multi-core chips, application developers have to develop parallel programs and deal with cumbersome issues such as synchronization tradeoffs and deadlock avoidance. In this setting, Transactional Memory (TM) has surfaced as a promising technique to simplify shared-memory parallel programming.

Recent years have seen the proposals of several different TM systems; however, most TM systems have been evaluated using microbenchmarks, which may not be representative of any real-world behavior, or individual applications, which do not stress a wide range of execution scenarios. To address this problem, I introduce the Stanford Transactional Applications for Multi-Processing (STAMP), the first comprehensive benchmark suite for evaluating TM systems. STAMP consists of eight benchmarks and represents several application domains, covers a wide range of transactional execution cases, and ports easily to many types of TM systems.

Using STAMP, I evaluate TM implementations based on hardware (HTM) and software (STM), and propose Signature-accelerated Transactional Memory (SigTM), a new hybrid TM system that combines the advantages of HTM and STM. SigTM uses small hardware signatures to accelerate the execution of software transactions, and thus presents a high-performance, flexible, and low-cost design. Moreover, SigTM is the first hybrid TM system to provide semantic guarantees between transactional and non-transactional code blocks.

Acknowledgements

Without a doubt, the completion of this dissertation would not have been possible without the help and support of my advisor, Christos Kozyrakis. It is now almost six years ago that I first met Christos. At the time, I was in the senior year of my undergraduate studies at The University of Texas at Austin, and Christos was finishing his first year as an assistant professor at Stanford. A few months earlier, I had applied to do my graduate studies in electrical engineering at Stanford, and Christos had decided to give me a phone call to tell me about all the great computer engineering research at Stanford. He convinced me to attend Stanford, and toward the end of my first year as a Stanford graduate student I joined one of his projects that was investigating innovative ways of simplifying parallel programming. Little did I realize what a great advisor I had chosen, and I am truly grateful for all the guidance and friendship that he has given me all these years.

Many other professors have also been instrumental in shaping my academic career. First, I would like to thank Kunle Olukotun, whose joint leadership of Stanford's Transactional Coherence and Consistency (TCC) research group with Christos has helped set the direction for my research. Along with Subhashish Mitra, Kunle and Christos have given me lots of constructive feedback that have been invaluable in helping me improve this dissertation. Finally, I would like to thank Yale Patt, whose introductory class for freshmen at The University of Texas at Austin inspired me to pursue a degree in computer engineering.

While doing graduate studies in computer at Stanford, many of my colleagues in the TCC research group have provided me with great assistance on numerous occasions. Of the TCC group members, I have collaborated with JaeWoong Chung

the most times, and I am truly inspired by his continuous ingenuity and positive attitude. Although, he was only part of the group for a year, Martin Trautmann made significant and amazing contributions to the early stages of my thesis research. My work has also relied greatly on the computer simulator infrastructure built by Austen McDonald and Jared Casper as well as the incredible "robot" automated experiment launcher from Brian Carlstrom. Last, I would like to express my gratitude to the administrative assistants for the TCC group, Teresa Lynn and Darlene Hadding.

Outside of the lab, my family and friends have been a continual source of encouragement and support. I am truly blessed to have Chanh and Ling Ling as my parents and Lyly as my sister, and I love all of them dearly. Although we are not related, I consider Adam Lee as my older brother, and he has enriched my life with both touches of epicurism as well as post-graduation employment. I thank my room mate of two years, Andrew Poon, and my good friend Joseph Koo, who even gave me a crash course on machine learning to help me with my thesis work. Finally, my graduate studies have been generously funded by Joseph and Honmai Goodman through the Stanford Graduate Fellowships program.

Thank you, and God bless.

Chí Cao Minh
September 2008

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**

1.1 The Difficulty of Parallel Programming 1
1.2 Transactional Memory 2
1.3 Contributions 4
1.4 Organization 5

2 Transactional Memory **7**

2.1 Transactional Memory Basics 7
2.2 Programming Transactional Memory 9
2.2.1 Implicit vs. Explicit Barriers 9
2.2.2 Weak vs. Strong Isolation 10
2.2.3 Nested Transactions 10
2.3 Transactional Memory System Taxonomy 11
2.3.1 Lazy vs. Eager Data Versioning 11
2.3.2 Optimistic vs. Pessimistic Conflict Detection 12
2.3.3 Hardware vs. Software 12
2.4 Hardware Transactional Memory 12
2.4.1 A Lazy Optimistic HTM 13
2.4.2 An Eager Pessimistic HTM 14
2.4.3 HTM Challenges 15

2.5 Software Transactional Memory 16
2.5.1 A Lazy Optimistic STM 16
2.5.2 An Eager Pessimistic STM 19
2.5.3 STM Challenges 22
2.6 Related Work 23
2.6.1 Hardware Transactional Memory 23
2.6.2 Software Transactional Memory 24

3 Benchmarking Transactional Memory **27**
3.1 Motivation and Requirements 27
3.2 Existing Benchmarks 28
3.2.1 Parallel Benchmarks 28
3.2.2 Transactional Memory Benchmarks 29
3.3 STAMP 30
3.3.1 Design Philosophy 31
3.3.2 Applications Overview 33
3.3.3 Implementation 33
3.4 STAMP Application Descriptions 34
3.4.1 `bayes` 34
3.4.2 `genome` 35
3.4.3 `intruder` 37
3.4.4 `kmeans` 40
3.4.5 `labyrinth` 41
3.4.6 `ssca2` 44
3.4.7 `vacation` 45
3.4.8 `yada` 47
3.5 Configurations and Data Sets 49

4 Characterizing and Applying STAMP **53**
4.1 Methodology 53
4.2 Basic Characterization 56
4.2.1 Transaction Length 56

4.2.2 Read and Write Set Sizes 58
4.2.3 Time in Transactions 58
4.2.4 Transaction Retries 59
4.2.5 Working Set Size 59
4.3 HTM and STM Performance Analysis 60
4.3.1 `bayes` 60
4.3.2 `genome` 60
4.3.3 `intruder` 62
4.3.4 `kmeans` 63
4.3.5 `labyrinth` 63
4.3.6 `ssca2` 64
4.3.7 `vacation` 64
4.3.8 `yada` 65
4.4 HTM and STM Performance Summary 65

5 Signature-accelerated Transactional Memory 67
5.1 HTM and STM Overhead 68
5.2 Strong Isolation 71
5.3 Signature-accelerated Transactional Memory 74
5.4 Hardware Signatures 75
5.5 Operation 78
5.6 Performance 81
5.7 Strong Isolation 82
5.8 Alternative Implementations 83
5.8.1 Lazy vs. Eager Data Versioning 83
5.8.2 Line vs. Object Granularity Conflict Detection 86
5.8.3 Broadcast Coherence vs. Directories 86
5.9 System Issues 87
5.9.1 Nesting 87
5.9.2 Hardware Multithreading 88
5.9.3 Thread Suspension and Migration 88

5.9.4 Paging 89
5.9.5 Inter-process Isolation 89
5.10 Related Work 90
5.10.1 Hybrid Transactional Memory 90
5.10.2 Signature-based HTMs 92
5.10.3 Strong Isolation 92

6 Evaluating Signature-accelerated Transactional Memory 95
6.1 Methodology 95
6.2 Performance Analysis 97
6.3 Hardware Cost Analysis 104
6.3.1 Read Signature Cost Analysis 104
6.3.2 Write Signature Cost Analysis 109
6.3.3 Cost Sensitivity to Number of Cores 110
6.3.4 Cost Analysis Summary 113

7 Conclusions 115
7.1 STAMP 115
7.2 SigTM 116

Bibliography 119

List of Tables

3.1 Publicly available benchmarks used to evaluate TM systems 32
3.2 The eight STAMP applications 33
3.3 STAMP's qualitative runtime transactional characteristics 50
3.4 Recommended configurations and data sets for STAMP 51

4.1 Configuration for the simulated multi-core system 54
4.2 Basic characterization of STAMP 57

5.1 User-level instructions for managing signatures in SigTM 76
5.2 Dynamic overhead for lazy STM and lazy SigTM barriers 81
5.3 Dynamic overhead for eager STM and eager SigTM barriers 85

6.1 Configuration for the simulated multi-core system with SigTM support 96

List of Figures

2.1 Example of an `atomic` block 8
2.2 Pseudocode for the basic functions in lazy STM 17
2.3 Pseudocode for the basic functions in lazy STM (continued) 18
2.4 Pseudocode for the basic functions in eager STM 20
2.5 Pseudocode for the basic functions in eager STM (continued) 21

3.1 Iteratively learning a Bayesian network 35
3.2 Pseudocode for `bayes` 36
3.3 Genome sequencing example 37
3.4 Pseudocode for `genome` 38
3.5 Pseudocode for `intruder` 39
3.6 K-means clustering example 40
3.7 Pseudocode for `kmeans` 41
3.8 Illustration of Lee's maze routing algorithm 42
3.9 Pseudocode for `labyrinth` 43
3.10 Pseudocode for `ssca2` 45
3.11 The three-tier design of `vacation` 46
3.12 Pseudocode for `vacation` 47
3.13 Delaunay mesh refinement example 48
3.14 Pseudocode for `yada` 49

4.1 STAMP speedups on HTM and STM 61

5.1 Overhead breakdown of the lazy STM 69

5.2 Overhead breakdown of the eager STM 70
5.3 Isolation and ordering violation cases for lazy STM 72
5.4 The code for the privatization scenario 73
5.5 Inserting an address into a signature 77
5.6 An example of operations on a Bloom filter 77
5.7 Pseudocode for the basic functions in lazy SigTM 79
5.8 Pseudocode for the basic functions in eager SigTM 84
5.9 Isolation and ordering violation cases for eager STM 85

6.1 STAMP speedups on lazy SigTM and lazy STM 99
6.2 STAMP speedups on eager SigTM and eager STM 100
6.3 Execution time breakdown of lazy SigTM and lazy STM 101
6.4 Execution time breakdown of eager SigTM and eager STM 102
6.5 Effect of read signature length on lazy SigTM performance 105
6.6 Effect of read signature length on eager SigTM performance 106
6.7 Effect of write signature length on lazy SigTM performance 107
6.8 Effect of write signature length on eager SigTM performance 108
6.9 Effect of read signature length on lazy SigTM scalability 111
6.10 Effect of read signature length on eager SigTM scalability 111
6.11 Effect of write signature length on lazy SigTM scalability 112
6.12 Effect of write signature length on eager SigTM scalability 112

Chapter 1

Introduction

Up till now, the performance of microprocessors has been continuously improving thanks to advances in manufacturing technologies. In recent years, however, conventional techniques for improving single-threaded performance have begun hitting fundamental limits. Designing increasingly deep and wide pipelines and increasing clock frequencies lead to undesirable levels of power consumption [2, 15]. Moreover, the traditional methods of exploiting instruction-level parallelism (such as out-of-order execution and speculation) in uniprocessors have been almost fully exhausted [95].

In response, processor manufacturers have shifted to multiprocessor designs [36, 50, 52, 63, 67, 93]. By using a smaller and simpler processor core design and replicating it many times on a single die, power requirements are reduced and a new opportunity for performance, in the form of thread-level parallelism, is exposed. The burden of achieving high performance has now been placed on software developers who must face the arduous task of writing parallel programs to take advantage of multiprocessor hardware.

1.1 The Difficulty of Parallel Programming

Writing programs to take advantage of thread-level parallelism on multiprocessors involves creating several parallel tasks that need to synchronize and communicate with each other. For shared memory systems, the coordination among parallel tasks

is commonly achieved via lock-based parallel programs. In this technique, locks are used to provide mutual exclusion for shared memory accesses that are used for communication among parallel tasks.

Unfortunately, parallel programming with locks is much more difficult than traditional sequential programming [47]. When programmers use locks, they must pick between two undesirable choices. The first option is to use coarse-grain locks, where large regions of code are indicated as critical sections. Adding coarse-grain locks to a program is relatively straightforward; however, coarse-grain locks may not permit high degrees of concurrency as instructions that could otherwise correctly execute concurrently may be serialized. The second option that programmers have is to use fine-grain locks, which entails placing lock so that the size of critical sections is minimized. Smaller critical sections permit greater concurrency, but the inherently greater number of locks in fine-grain schemes leads to higher complexity, which may not even result in better performance. The higher program complexity also makes it more likely for the code to have problems such as deadlock, convoying, or priority inversion.

1.2 Transactional Memory

Transactional Memory (TM) [47] was created to simplify parallel programming and relieve software developers from the difficulties associated with lock-based parallel programming. With locks, programmers need to explicitly specify and manage the synchronization among threads; however, with TM, programmers simply mark code segments as transactions that should execute *atomically* and *in isolation* with respect to other code, and the TM system manages the concurrency control for them.

All TM systems use either hardware-based or software-based approaches to implement the two basic TM mechanisms: data versioning and conflict detection. Hardware TM (HTM) systems use hardware caches for data versioning and hook into cache coherence protocols for conflict detection [11, 41, 65, 72]. Since the processor can transparently track loads and stores for the transactional bookkeeping, HTMs have low transactional overhead; however, the complexity and cost of redesigning

the caches and coherence protocols to support TM can be significant. In contrast, Software TM (STM) adds instrumentation code (read and write barriers) to interact with an STM software library [32, 43, 46, 59, 80]. Since STMs are software-based, they are more cost-effective and flexible than HTMs; however, the software barriers and libraries incur much greater overhead than hardware-based solutions. Ideally, a TM system should be a hybrid combination of hardware and software to combine the best of HTM and STM designs.

Another important difference between HTM and STM is in their semantics. To execute all code sequences predictably, a TM system must support *strong isolation*, which guarantees that code in transactions runs isolated from both transactional and non-transactional code. HTMs naturally have strong isolation as the processor sees all memory accesses. On the other hand, STMs must add extra instrumentation code on non-transactional loads and stores to provide strong isolation. Since this is an additional source of overhead, high-performance STMs usually sacrifice strong isolation, which allows some unpredictable code execution scenarios. A better solution, however, would be to add strong isolation to STM by using hardware. This would present a solution that retains the cost-effectiveness and flexibility of STMs while offering predictable semantics and high performance.

Although many HTM and STM systems have been proposed, the tools and workloads are still missing to analyze and compare the proposals. Most TM systems have been evaluated using either microbenchmarks or single applications. Unfortunately, microbenchmarks may not be representative of any real-world behavior, and individual applications do not stress a wide range of execution scenarios. To provide a thorough analysis of TM systems, a benchmark suite that covers a wide range of transactional scenarios (e.g., varying sizes of read and write sets, differing amounts of contention, etc.) is needed. With these workloads, the TM community can better understand the tradeoffs between HTM and STM and design a hybrid TM system that combines the advantages of each.

1.3 Contributions

In this dissertation, I present the design of a new effective hybrid transactional memory system. Specifically, I:

- Create the *Stanford Transactional Applications for Multi-Processing (STAMP)*, a new benchmark suite for transactional memory systems. I describe the algorithms and data structures, the parallelization strategy, and the use of transactions in each of the eight STAMP applications.

- Provide a transactional characterization of the STAMP applications. In particular, I measure the transaction length, the sizes of the read and write sets, the amount of time spent in transactions, and the average number of retries per transaction. This analysis quantitatively demonstrates that STAMP covers a wide range of transactional scenarios.

- Demonstrate the usefulness of STAMP for TM research by using it to analyze the performance of two different HTMs and two different STMs. The conventional wisdom in the TM community has been that eager versioning leads to better performance than lazy versioning and increasing amounts of hardware support for TM lead to significant performance improvements. Using STAMP, I show that these generalizations are invalid in certain cases and discuss the technical issues behind them. With STAMP, I analyze the tradeoffs between HTM and STM, and motivate a hybrid hardware and software TM design.

- Describe the hardware and software components of SigTM, my new hybrid transactional memory system that uses hardware signatures for read set and write set tracking. SigTM improves the performance of software transactions while providing strong isolation guarantees.

- Analyze the SigTM design by using STAMP. I show that SigTM effectively accelerates software transactions and approaches the performance of HTMs at a much lower hardware cost.

1.4 Organization

The remainder of this dissertation is organized as follows. Chapter 2 introduces Transactional Memory, explains basic TM concepts, and highlights some TM designs. In Chapter 3, I explain why previous approaches to evaluating TM have been inadequate, and I introduce the STAMP benchmark suite to address this problem. A quantitative analysis of STAMP then follows in Chapter 4. Chapter 5 introduces SigTM, my new hybrid TM design that combines the high performance and predictable semantics of HTMs with the flexibility and cost-effectiveness of STMs. Next, Chapter 6 uses STAMP to present a quantitative analysis of the SigTM design. Finally, Chapter 7 concludes this dissertation.

Chapter 2

Transactional Memory

Multi-core chips are now commonplace in server, desktop, and even embedded systems. However, multi-core chips create an inflection point for mainstream software development. To benefit from the increasing number of cores per chip, application developers have to create parallel programs and deal with cumbersome issues such as synchronization tradeoffs, deadlock avoidance, and races. In this setting, Transactional Memory (TM) [47] has surfaced as a promising technique to help with shared-memory parallel programs.

2.1 Transactional Memory Basics

With conventional multithreaded programming on shared memory machines, programmers need to manually manage the synchronization among threads when they use locks. For example, they must select the lock granularity, create an association between shared data and locks, and manage lock contention. In other words, with locks, programmers not only need to declare where synchronization is used, they must also implement how synchronization occurs.

In contrast, with Transactional Memory (TM), programmers simply declare where synchronization occurs, and the TM systems handles the implementation. In more detail, with TM, programmers indicate that a code segment should be executed as a *transaction* by placing that group of instructions in an `atomic` block as shown in

```
atomic {
    if (x != NULL) x.foo();
    y++;
}
```

Figure 2.1: Groups of instructions are executed as a transaction by placing them inside an `atomic` block.

Figure 2.1. It is the responsibility of the TM system to guarantee that the transactions have the following properties: atomicity, isolation, and serializability. First, with *atomicity*, either all or none of the instructions in the transaction must appear to occur. Next, having *isolation* means that none of the intermediate state of a transaction is visible outside of the transaction. Finally, *serializability* requires that the execution order of concurrent transactions is equivalent to some sequential execution order of the same transactions.

The way that TM systems achieve good parallel performance is by providing *optimistic concurrency control*. When the TM system executes the body of an `atomic` block, it does so speculatively (hence the name "optimistic"). While the body is executed, any memory addresses that are read are added to a *read set*, and ones that are written are added to a *write set*. The write set is also used to provide *data versioning* by keeping track of either the old or new value for the written address. Finally, at the end of the `atomic` block, the TM system ends, or *commits*, the transaction.

To verify that the speculative execution of the transaction is valid, the TM system compares the read and write sets of all concurrent transactions. This allows it to perform fine-grain read-write and write-write conflict detection. If no conflicts are detected, the transaction commits successfully, otherwise it is *aborted*, and execution is *rolled back* to the beginning of the `atomic` block and retried.

The key idea in Transactional Memory is that because of their atomicity, isolation, and serializability properties, transactions can be used to build parallel programs. Using large `atomic` blocks simplifies parallel programming because it provides ease-of-use and good performance. First, like coarse-grain locks, it is relatively easy to reason about the correctness of transactions. Second, to achieve performance comparable to

that of fine-grain locks, the programmer does not have to do any extra work because the TM system will automatically handle that task.

Two more advantages that transactions have over locks are that they provide *composability* and *failure atomicity*. With locks, two code segments that are correct individually may be incorrect when they are combined. For example, consider a hash table that uses locks to provide thread-safe insert and remove operations. Unfortunately, combining these two operations to move an item between two different hash tables may not be correct as this *composite* operation exposes an invalid intermediate state where the item does not exist in either hash table. In contrast, simply enclosing two transactions by a larger transaction guarantees atomicity. Moreover, unlike locks, correctness is ensured even in the presence of failures. For example, transactions do not suffer from situations where a thread holds a lock, subsequently dies, and prevents the progress of any other threads that wish to acquire that same lock.

2.2 Programming Transactional Memory

When implementing a TM system, there are many choices in how transactions interact with other programming abstractions and in what their semantics should be. In this section, I will introduce two of these choices: implicit versus explicit barriers and weak versus strong isolation. A more through coverage of these design choices and others can be found in [18, 21, 56]. I will also briefly describe nested transactions.

2.2.1 Implicit vs. Explicit Barriers

Memory addresses can be added to read and write sets either implicitly or explicitly. To do so implicitly requires either compiler or hardware support to find all the memory accesses within all the `atomic` blocks. Without this support, programmers have to explicitly add barriers to their code to annotate memory accesses to shared variables. Implicit barriers have the advantage of allowing programmers to compose existing non-transactional code to create transactions. However, with explicit barriers, an expert programmer may be able to perform optimizations that lead to smaller read

and write sets than those achievable with implicit barriers. For example, barriers for local and private variables can be optimized away.

2.2.2 Weak vs. Strong Isolation

The terms *weak isolation* and *strong isolation* were first introduced in [56] (though the same concepts first appeared in [12], but as *weak atomicity* and *strong atomicity*). The distinction between *weak isolation* and *strong isolation* is that the former guarantees transactional semantics only among `atomic` blocks. In comparison, *strong isolation* also guarantees transactional semantics between transactional and non-transactional code (in addition to among just transactional code blocks). Essentially, with strong isolation all non-transactional instructions effectively execute as one-instruction `atomic` blocks.

From the programmer's point of view, strong isolation makes it easier to reason about correctness, especially when shared data becomes private (privatization) or when private data is transferred into a shared domain (publication). However, in spite of the advantages of strong isolation, some TM implementations may choose to implement weak isolation as it can lead to a design with higher performance. The subject of weak versus strong isolation will be revisited later in more detail in Section 5.2.

2.2.3 Nested Transactions

Nested transactions occur when the dynamic extent of a transaction is fully enclosed by that of another transaction. For example, nested transactions can arise when an application programmer writes a transaction that contains a library call that executes transactions itself.

There are three kinds of nested transactions: flattened, closed, and open. With *flattened nested* transactions, any nested transaction boundary annotations are simply ignored and aborting the child transaction causes the parent transaction to also abort. On the other hand, with *closed nested* transactions, aborting the child transaction only causes re-execution of the child transaction. With both flattened and closed nested

transactions, modifications made by the child transaction are only externally visible after the parent transaction commits. In contrast, *open nested* transactions behave like closed nested transactions, except that the changes of the child transaction are externally visible as soon as it commits. Because of this, to properly roll back an open nested transaction, compensation actions must be executed after aborting a transaction. Longer discussions of nested transactions can be found in [18, 61].

2.3 Transactional Memory System Taxonomy

There are many design decisions that can be made when creating a TM system. In this section, I will introduce three categories that can be used to classify TM systems: the scheme used for *data versioning*, the policy used for *conflict detection*, and whether a *hardware-based* or *software-based* approach is used. A more thorough description of the taxonomy of TM systems can be found in [56]. Deciding which design choice is the best requires a comprehensive set of applications that covers a wide variety of transactional program characteristics, such as large and small sizes of read and write sets. Approaches used to evaluating different TM designs will be covered later in Chapter 3.

2.3.1 Lazy vs. Eager Data Versioning

When a transaction performs a write access to memory, the update to the memory address can be performed either lazily or eagerly. In *lazy data versioning*, updates to memory are deferred by buffering them on the side. When a transaction reaches its commit point, the lazy TM system then updates memory by applying all the deferred updates in its data versioning buffer. On the other hand, *eager data versioning* applies memory updates directly to memory after recording the old value in an undo log. If a transaction aborts in an eager TM system, the undo log is used to revert all the updates made by the transaction.

The type of data versioning scheme used gives a TM system different advantages and disadvantages. With lazy data versioning, transaction aborts are fast because

main memory does not contain any speculative updates from the transaction. In contrast, eager data versioning schemes pay a performance penalty on transaction abort as they must process their undo logs. However, lazy schemes have slower transaction commits as they defer all transactional updates till the commit point. Finally, for software-based TMs, lazy versioning leads to slower read accesses than eager versioning. This is because the software data versioning buffer may contain values newer than those in main memory, necessitating a search of the buffer during each speculative read access.

2.3.2 Optimistic vs. Pessimistic Conflict Detection

TM systems can take either an optimistic or pessimistic approach to performing conflict detection. With *optimistic conflict detection*, a TM system optimistically assumes that a transaction will commit successfully and only performs conflict detection late at the end of the transaction. On the other hand, *pessimistic conflict detection* checks for conflicts on each memory access during transaction execution. Since conflict detection is performed early, less work is potentially wasted if a transaction aborts; however, optimistic conflict detection allows more transaction interleavings to successfully commit and guarantees forward progress. Finally, lazy data versioning is usually combined with optimistic conflict detection and eager data versioning with pessimistic conflict detection, as these are the synergistic combinations.

2.3.3 Hardware vs. Software

All TM systems must implement data versioning and conflict detection; however, these tasks can be implemented in either hardware (HTM) or software (STM).

2.4 Hardware Transactional Memory

Hardware Transactional Memory (HTM) systems implement data versioning and conflict detection by enhancing both the caches and the cache coherence protocol in a

multi-core system [11, 14, 41, 65, 72]. Sections 2.4.1 and 2.4.2 describe an HTM implementation with lazy data versioning and optimistic conflict detection and an HTM with eager data versioning and pessimistic conflict detection, respectively.

2.4.1 A Lazy Optimistic HTM

The lazy optimistic HTM that I will describe is similar in design to the TCC system [41]. Both use the cache to buffer the write set until the transaction commits, and conflict detection is implemented using coherence messages when one transaction attempts to commit. However, my lazy optimistic HTM has two important differences from the TCC design. First, TCC executes all user code in transactional mode, while my lazy optimistic HTM uses transactional mechanisms only for user-defined transactions. The rest of the code executes as a conventional multithreaded program with MESI cache coherence and sequential consistency. Second, TCC uses a write-through coherence protocol that provides conflict resolution at word granularity. My lazy optimistic HTM uses write-back caches, which require conflict detection at cache line granularity.

The lazy optimistic HTM extends each cache line with one *read bit (R)* and one *write bit (W)* that indicate membership in a transaction's read set and write set, respectively. A transaction starts by taking a register checkpoint using a hardware mechanism. A store writes to the cache and sets the W bit. If there is a cache miss, the cache line is requested in the shared state. If there is a hit but the line contains modified data produced prior to the current transaction (modified and W bit not set), it first writes the data back to lower levels of the memory hierarchy. A load reads the corresponding word and sets the R bit if the W bit is not already set. If there is a cache miss for the load, the line is retrieved in the shared state as well. Overflows of the cache are handled by acquiring a unique hardware lock to temporarily serialize the execution of transactions.

When a transaction completes, it arbitrates for permission to commit by acquiring a unique hardware lock. This implies that only a single transaction may be committing at any point in time. Parallel commit can be supported by using a two-phase

protocol [24]. Next, the lazy optimistic HTM generates snooping messages to request exclusive access for any lines in its write set (W bit set) that are in shared state. At this point, the transaction is validated. Finally, it commits the write set atomically by flash resetting all R and W bits and then releasing the commit lock. All data in the write set are now modified in the processor's cache but are non-transactional and can be read by other processors.

An ongoing transaction detects a conflict when it receives a coherence message with an exclusive request for data in its read or write set. Such a message can be generated by a committing transaction or by a non-transactional store. A violated transaction is rolled back by flash invalidating all lines in the write set, flash resetting all R and W bits, and restoring the register checkpoint. Since memory accesses both inside and outside transactions can generate coherence messages, the lazy optimistic HTM's conflict detection mechanism provides strong isolation.

Transaction starvation is avoided by allowing a transaction that has been retried multiple times to acquire the commit lock at its very beginning. Forward progress is guaranteed because a validated transaction cannot abort. To guarantee this, a validated transaction sends negative acknowledgments (NACKs) to all types of coherence requests for data in its write set and exclusive requests for data in its read set. Once validated, an HTM transaction must execute just a few instructions to flash reset its W and R bits, hence the window of NACKs is extremely short. If a transaction receives a shared request for a line in its read set or write set prior to reaching the commit stage, it responds that it has a shared copy of the line and downgrades its cache line to the shared state if needed.

2.4.2 An Eager Pessimistic HTM

The eager pessimistic HTM described in this section is based on the LogTM system [65]. As in the lazy optimistic HTM, the cache in the eager pessimistic HTM is modified to have one read bit and one write bit per cache line, and the bits are used in the same manner in both HTMs. The eager pessimistic HTM also has hardware to

accelerate manipulation of an undo log. Writes to memory addresses inside transactions update memory directly after the old value and its virtual address are recorded in the undo log. To prevent redundant log operations, the W bits in the cache are used as a filter. When a transaction aborts, a software handler walks the software log to restore the old values to memory. Transaction commit is very fast because no values need to be copied to memory and the pointer to the log simply needs to be reset.

By using the R and W bits in the cache and the coherence protocol, conflicts among transactions are detected and strong isolation is provided. When a processor receives a coherence message with a request for something in its read set or write set (detected by checking the cache's R and W bits), it responds with either an ACK (no conflict) or a NACK (conflict). The requesting processor then receives this response and resolves the conflict. If an overflow of the cache occurs, the event is recorded and conflicts are conservatively indicated.

Thus far, the operation of the eager pessimistic HTM has been the same as LogTM; however, my eager pessimistic HTM does have a few important differences. First, it uses broadcast coherence with a MESI protocol. Since LogTM uses directories with a MOESI protocol, it introduces "sticky" states to delay modifications to the directories so that coherence messages are still received for lines that are overflowed. Second, my eager pessimistic HTM uses a Bloom filter [9] to record overflowed addresses instead of a single bit. While, this technique still detects conflicts conservatively, it admits much fewer false conflicts than a single bit (at a slightly increased hardware cost). Finally, to guarantee forward progress, a transaction that aborts several times in my eager pessimistic HTM acquires a unique global hardware lock to guarantee that it commits successfully.

2.4.3 HTM Challenges

Because HTMs use hardware to implement data versioning and conflict detection, they offer high performance and inherently provide strong isolation. High performance is achieved because no software annotations are required on memory accesses,

and strong isolation is naturally provided by leveraging the cache coherence protocol for conflict detection. Unfortunately, implementing all transactional features in hardware is both costly and inflexible. All the lines in the cache must be modified, and extra hardware may need to be added to support an undo log. This hardware cost is further exacerbated if transaction nesting is supported by hardware. Furthermore, because hardware resources are finite and shared by all threads, HTMs face transaction virtualization challenges. For example, HTMs can encounter difficulties when transactionally-accessed lines overflow the capacity of the cache or when transaction metadata in the hardware must survive events like context switches. Finally, HTMs that rely on coherence protocols such as MESI cannot detect conflicts at finer granularity than lines. This can lead to false sharing among threads if they access disjoint words in the same line, and the false conflicts that result hurt performance.

2.5 Software Transactional Memory

STM systems implement version management and conflict detection using software-only techniques [32, 43, 46, 59, 80]. Sections 2.5.1 and 2.5.2 describe an STM implementation with lazy data versioning and optimistic conflict detection and an STM with eager data versioning and pessimistic conflict detection, respectively.

2.5.1 A Lazy Optimistic STM

In this section, I will describe a lazy optimistic STM based on the TL2 STM [32]. TL2 is a lock-based STM that implements optimistic concurrency control for both read and write accesses and scales well across a range of contention scenarios [34]. It is the software equivalent of the lazy optimistic HTM from Section 2.4.1. The source code for the lazy optimistic STM is open source and can be downloaded from `http://stamp.stanford.edu`.

Figures 2.2 and 2.3 provide a simplified overview of the lazy optimistic STM for a C-style programming language. Refer to [32] for a discussion of TL2 for object-oriented languages. The lazy optimistic STM maintains a global version clock used

```
1: procedure LazySTMtxStart
2:     checkpoint()
3:     ReadVersion ← GlobalClock
4: end procedure

5: procedure LazySTMreadBarrier(addr)
6:     if bloomFilter.member(addr) and writeSet.member(addr) then
7:         return writeSet.lookup(addr)
8:     end if
9:     value ← Memory[addr]
10:    if locked(addr) or (timeStamp(addr) > ReadVersion) then
11:        conflict()
12:    end if
13:    readSet.insert(addr)
14:    return value
15: end procedure

16: procedure LazySTMwriteBarrier(addr, data)
17:     bloomFilter.insert(addr)
18:     writeSet.insert(addr, data)
19: end procedure
```

Figure 2.2: Pseudocode for the basic functions in lazy STM.

to generate time stamps for all data. To implement conflict detection at word granularity, it also associates a lock with every word in memory by using a hash function. The first bit in the lock word indicates if the corresponding word is currently locked. The remaining bits are used to store the time stamp generated by the last transaction to write the corresponding data.

A transaction starts (`LazySTMtxStart`) by using `setjmp` to take a checkpoint of the current execution and by reading the current value of the global clock into the variable $ReadVersion$. A transaction updates a word by using a write barrier (`LazySTMwriteBarrier`). The barrier first checks for a conflict with other committing or committed transactions by using the corresponding lock for the address to be written. A conflict is signaled if the word is locked or its time stamp is higher than the value in $ReadVersion$. Assuming no conflict, the store address and data are added

```
1: procedure LazySTMtxCommit
2:     for all addr in writeSet do
3:         if lock(addr) fails then
4:             conflict()
5:         end if
6:     end for
7:     WriteVersion ← Fetch&Increment(GlobalClock)
8:     for all addr in readSet do
9:         if locked(addr) or (timeStamp(addr) > ReadVersion) then
10:            conflict()
11:        end if
12:    end for
13:    for all addr in writeSet do
14:        Memory[addr] ← writeSet.lookup(addr)
15:    end for
16:    for all addr in writeSet do
17:        timeStamp(addr) ← WriteVersion
18:        unlock(addr)
19:    end for
20: end procedure

21: procedure LazySTMtxAbort
22:    readSet.reset()
23:    writeSet.reset()
24:    doContentionManagement()
25:    restoreCheckpoint()
26: end procedure
```

Figure 2.3: Pseudocode for the basic functions in lazy STM (continued).

to the *write set*, a hash table that buffers the transaction output until it commits. The STM also maintains a software 32-bit Bloom filter [9] for the addresses in the write set. A transaction loads a word using a read barrier (`LazySTMreadBarrier`). The barrier first checks if the latest value of the word is available in the write set and uses the Bloom filter to reduce the number of hash table lookups. If the address is not in the write set, it checks for a conflict with other committing or committed transactions. Assuming no conflict, it inserts the address to the *read set*, a simple FIFO that tracks read addresses. Finally, it loads the word from memory and returns its value to the user code.

In order to commit its work (`LazySTMtxCommit`), a transaction first attempts to acquire the locks for all words in its write set. If it fails on any lock then a conflict is signaled. Next, it atomically increments the global version clock by using the atomic instructions in the underlying ISA (e.g., `cmpxchg` in x86). It also validates all addresses in the read set by verifying that they are unlocked and that their time stamp is not greater than $ReadVersion$. At this point, the transaction is *validated* and guaranteed to complete successfully. The final step is to scan the write set twice in order to copy the new values to memory, update their time stamp to $WriteVersion$, and release the corresponding lock.

The lazy optimistic STM handles conflicts in the following manner. If a transaction fails to acquire a lock while committing, it first spins for a limited time and then aborts by using `LazySTMtxAbort`. To provide liveness, the STM retries the transaction after a random backoff delay that is linearly biased by the number of aborts thus far.

2.5.2 An Eager Pessimistic STM

To create an eager pessimistic STM, I modified the lazy optimistic STM from Section 2.5.1. Figures 2.4 and 2.5 provide pseudocode that show the basic functions of the eager pessimistic STM. Like the lazy optimistic STM, a global version clock is used and all memory addresses are associated with a lock word via a hash function.

```
 1: procedure EagerSTMtxStart
 2:     checkpoint()
 3:     ReadVersion ← GlobalClock
 4: end procedure

 5: procedure EagerSTMreadBarrier(addr)
 6:     value ← Memory[addr]
 7:     if locked(addr) or (timeStamp(addr) > ReadVersion) then
 8:         conflict()
 9:     end if
10:     readSet.insert(addr)
11:     return value
12: end procedure

13: procedure EagerSTMwriteBarrier(addr, data)
14:     if lock(addr) fails then
15:         conflict()
16:     end if
17:     writeSet.insert(addr, Memory[addr])
18:     Memory[addr] ← data
19: end procedure
```

Figure 2.4: Pseudocode for the basic functions in eager STM.

Starting a transaction (`EagerSTMtxStart`) performs the same tasks as in the lazy optimistic STM equivalent.

Since this STM updates values eagerly, the code for the read and write barriers differs from that of the lazy optimistic STM. Because main memory contains the most recently updated values, `EagerSTMreadBarrier` does not need to perform a scan of the write set, which also makes it unnecessary to have a Bloom filter. To perform conflict detection pessimistically, `EagerSTMwriteBarrier` first attempts to acquire a lock for the address being written. If this lock acquisition fails, a conflict is signaled, otherwise the original value of the address is saved in the undo log and main memory is updated.

Committing a transaction in the eager pessimistic STM involves several steps. Because locks have been acquired incrementally by the write barriers, `EagerSTMtxCommit`

```
1:  procedure EagerSTMtxCommit
2:      WriteVersion ← Fetch&Increment(GlobalClock)
3:      for all addr in readSet do
4:          if locked(addr) or (timeStamp(addr) > ReadVersion) then
5:              conflict()
6:          end if
7:      end for
8:      for all addr in writeSet do
9:          timeStamp(addr) ← WriteVersion
10:         unlock(addr)
11:     end for
12: end procedure

13: procedure EagerSTMtxAbort
14:     readSet.reset()
15:     for all addr in writeSet do
16:         Memory[addr] ← writeSet.lookup(addr)
17:     end for
18:     for all addr in writeSet do
19:         unlock(addr)
20:     end for
21:     writeSet.reset()
22:     doContentionManagement()
23:     restoreCheckpoint()
24: end procedure
```

Figure 2.5: Pseudocode for the basic functions in eager STM (continued).

does not need to perform a pass over the write set to lock all the written addresses. Then, as in the lazy optimistic STM, the $WriteVersion$ is incremented and the read set is validated. If the read set passes validation, the transaction can successfully commit. The most recent values are already in memory, so no copying is required. Commit ends by doing a final pass over the write set to update the version numbers and release the locks protecting the write set.

Although transaction commit is faster in the eager pessimistic STM than in its lazy optimistic equivalent, aborting transaction can be much slower. After resetting the read set, `EagerSTMtxAbort` must iterate over the undo log to revert the contents of memory. Atomicity of the value restoration is provided by the locks acquired in `EagerSTMwriteBarrier` for each of the written addresses. Then, after the values are restored, another traversal must be performed over the write set to release all of the locks. Finally, the write set is reset and a randomized linear backoff biased by the current number of aborts is performed to prevent livelock.

2.5.3 STM Challenges

As STMs use software to implement data versioning and conflict detection they are relatively inexpensive build and can also easily be changed to implement a variety of transactional policies. However, doing everything in software incurs much greater overhead than if dedicated hardware were used. For example, software barriers must be used to annotate memory accesses and data structures must be maintained and queried to perform conflict detection. Moreover, in order prevent further performance degradation, many STMs only support weak isolation of transactions. Providing strong isolation in STMs can be achieved by adding read and write barriers for memory accesses outside of transactions, which exacerbates the software overhead of STM systems. Finally, the presence of STM library code can increase cache pressure, resulting in more cache misses.

2.6 Related Work

This section briefly overviews some of the work done in designing HTM and STM systems. Larus and Rajwar present more detailed coverage of the history of TM in [56].

2.6.1 Hardware Transactional Memory

Hardware Transactional Memory first appeared when Herlihy and Moss proposed using HTM for building lock-free data structures [47]. Another early HTM system was Speculative Lock Elision (SLE) [69]. By using hardware support, SLE optimistically executed lock-demarcated critical regions as transactions. This allowed programmers to conservatively use frequent lock-based synchronization, while achieving the performance of more well-tuned locks. Transactional Lock Removal (TLR) [70] was a later extension to SLE that used time stamps for conflict resolution to provide transactional semantics and prevent starvation.

Later HTM systems focused on addressing the limited transaction size of early HTM designs. In the Transactional Coherence and Consistency (TCC) system [41], a new shared memory model is presented where all code executes transactionally. Transactional read and writes are buffered in the cache, and if an overflow occurs, TCC temporarily enters a nonspeculative mode. UTM and LTM [3] use a local uncached memory region as extra storage for cache overflows, and LogTM [65] modifies the coherence protocol to allow transactional state to escape the cache.

More recently, HTM designs have solved the problem of virtualizing transactional state across time. The VTM system [71] accomplished this by placing a data structure (the XADT) in virtual memory. When space or time virtualization is required, the XADT is used to hold transactionally-accessed cache lines. To accelerate processing of the XADT, VTM adds dedicated hardware. In contrast, the PTM [27] and XTM [29] systems utilize pages from the virtual memory system to virtualize transactional state at lower hardware costs than VTM.

Two recent HTMs have improved hardware support for transactions of unbounded size. OneTM [10] features simple hardware that allows unbounded transaction sizes,

but with the restriction that only one unbounded transaction can be executing at any given time. The execution of all non-overflowed transactions, however, is unhindered in the presence of an overflowed transaction. In TokenTM [14], the limit of a single overflowed transaction is removed while still retaining relatively simple hardware. This is accomplished by associating tokens with each memory block and using them to perform precise and unbounded transactional conflict detection. Like OneTM, non-overflowed transactions in TokenTM are not affected by overflowed transactions.

Finally, some HTM proposals have looked at incorporating hardware support for software. In MetaTM [73], architectural support is added for running a transactional operating system (TxLinux [77]). The FlexTM system [87] shows how transactional hardware can be decoupled into four components: access tracking, conflict tracking, data versioning support, and conflict notification. Having these well-defined components, makes it potentially easier to reuse transactional hardware for non-transactional purposes.

2.6.2 Software Transactional Memory

The first STM design was proposed by Shavit and Touitou [85] and required programmers to statically specify each transaction's data versioning requirements. Synchronization within the STM itself was accomplished with non-blocking (lock-free) techniques. STM systems then transitioned to dynamic data versioning designs with the DSTM system [46]. Next, the WSTM system [43] showed how to integrate STM into programming languages, giving birth to modern STM designs. Later work also showed how to handle exceptions and side-effects in transactions [42] and how to integrate transactional memory with modern programming languages [18–20, 60].

Many STMs have explored different techniques for data versioning, conflict detection, and contention management. Using the DSTM system, Scherer and Scott explored a variety of contention management schemes for STM [83]. Other work on comparing contention management studies include [37, 38]. Based on this work, the ASTM system [58] provided a system that could adapt its contention management policy to enable the best performance for a given workload. In [80], Saha et al. present

the McRT STM and provide a comparison between lazy and eager data versioning, word-granular and object-granular locks, and whether readers or writers should be preferred when acquiring locks. The proposal of the McRT STM also marked a transition from non-blocking to blocking designs in STMs. Finally, the Transactional Locking (TL) STM [33], provided an analysis of optimistic versus pessimistic conflict detection.

Two recent STM designs incorporated time stamps to efficiently check the consistency of data accessed in transactions. The first of these was TL2 [32], which used a single global counter and a deferred-update and blocking implementation. A scalable replacement for the single global counter was then later presented in conjunction with the LSA STM, which introduced an efficient method for verifying the consistency of transactional objects on each access [75, 76].

Finally, much work has been done in using compilers to support software transactions. A thorough coverage of several compiler techniques to optimize STMs is presented in [1]. Later work also showed how to use compilers in managed languages to minimize the extra overhead associated with providing strong isolation for STMs [86].

Chapter 3

Benchmarking Transactional Memory

Benchmarks are software programs that are used to evaluate certain aspects of computers and help guide the design of future systems. However, the conclusions that are drawn from benchmark results are limited by the ability of the benchmarks to cover relevant aspects of the computer system. In this chapter, I argue that current approaches to benchmarking Transactional Memory systems have been inadequate. As a solution, I propose the *Stanford Transactional Applications for Multi-Processing (STAMP)*, a new benchmark suite designed specifically for comprehensively analyzing Transactional Memory systems.

3.1 Motivation and Requirements

Most evaluations of Transactional Memory systems have relied on microbenchmarks or parallel applications from benchmark suites like SPEComp [91] or SPLASH-2 [97]. While microbenchmarks are useful in targeting specific system features, they are not representative of how full applications will behave on TM systems. On the other hand, the benchmarks in SPEComp and SPLASH-2 are full applications, but they have been heavily optimized by an expert (typically as part of a computer science Ph.D. thesis) to minimize synchronization and communication across threads. Thus,

converting their fine-grain lock-protected regions to transactions leads to programs that rarely use transactions [30], making it hard to evaluate the differences among TM systems. Furthermore, this behavior is not necessarily representative of how mainstream programmers will use transactions in their programs. The most appealing potential of TM for many programmers is the ability to write simple parallel code with frequent use of coarse-grain transactions that performs as well as code that has been carefully optimized to use fine-grain locks.

For a benchmark suite to enable a thorough analysis of a wide range of TM systems, it must have three key features. First, it must target a *variety of algorithms and application domains that can benefit from TM*. Second, it must cover a *wide range of transactional characteristics*, such as a range of transaction lengths or sizes of read and write sets. Moreover, the amount of time spent in transactions should be varied. Certain applications or data sets in the suite should generate cases where a significant portion of execution time is spent in transactions. This requirement allows the performance of TM systems to be evaluated with frequently-used, coarse-grain transactions. Finally, the benchmark suite must be *compatible with a large number of TM systems*, covering hardware, software, and hybrid designs.

3.2 Existing Benchmarks

Researchers and industry consortiums have created many benchmarks for evaluating parallel systems. Recently, some benchmarks for TM systems have emerged as well. This section reviews the scope, strengths, and shortcomings of these efforts with respect to evaluating TM systems.

3.2.1 Parallel Benchmarks

Three of the oldest and most widely used parallel benchmarks are SPLASH-2 [97], NPB OpenMP [49], and SPEComp [91]. More recently, domain-specific parallel benchmarks have also been created, such as BioParallel [48] for bioinformatics and

MineBench [66] for data mining. These benchmark suites consist of several applications that span a variety of algorithms. Additionally, all of them utilize OpenMP as their programming model, except for SPLASH-2, which uses a Pthreads-like model through the use of ANL macros.

PARSEC [8] is a very recent parallel benchmark that was created to address some of the shortcomings in the above-mentioned benchmarks. In selecting the PARSEC applications, care was taken to use state-of-art techniques in a range of application domains not limited to just high-performance computing. The applications are written in either C or C++ and use either OpenMP or Pthreads to implement the parallelization.

Even though these benchmarks are useful in analyzing parallel systems in general, their applicability to TM systems is limited. For example, many of the applications in these benchmarks consist of regular algorithms that can be parallelized without stressing synchronization (i.e., just using barriers is sufficient). Furthermore, in order to achieve the highest possible performance, the irregular applications in these benchmarks have been carefully optimized by expert programmers to minimize the time spent in critical regions. Converting the critical regions in these applications to transactions [27, 31, 62, 65], leads to programs with small transactions that are rarely used. Consequently, these benchmarks are unable to sufficiently stress the underlying TM system and do not generate most of the interesting cases of transactional behavior with respect to transaction length and frequency or sizes of read and write sets. Since manual tuning of the application code by an expert contradicts the primary goal of transactional memory (practical parallel programming for mainstream developers), usage of these benchmarks alone to evaluate TM systems may be misleading.

3.2.2 Transactional Memory Benchmarks

To better target TM systems, researchers have begun creating new workloads. Existing efforts can be grouped into two general categories: microbenchmarks and individual applications. TM microbenchmarks are typically composed of transactions that execute a few operations on a data structure like a hash table or red-black

tree [31, 32, 55, 59, 80]. These microbenchmarks are easy to develop, parameterize, and port across systems. Furthermore, they are very useful for isolating particular cases of interest in the TM system. However, they are not representative of full applications. Full applications are likely to have transactions that consist of many operations across several data structures and may also include significant amounts of parallel or sequential work between transactions. Thus, the transactional characteristics of microbenchmarks and full applications may be significantly different.

A few larger workloads that target TM systems have been designed to address the disparity between microbenchmarks and full applications: Delaunay mesh generation [84], database management [39], BerkeleyDB [31], maze routing [96], and Delaunay mesh refinement and agglomerative clustering [54]. These applications represent realistic workloads and avoid the pitfalls of microbenchmarks. However, as they are all standalone and not part of a larger suite, their coverage of algorithms and transactional behaviors is limited. Moreover, only the first two of these applications are publicly available to the research community.

More recent work has focused on the creation of TM benchmark suites. Perfumo et al. [68] have created a suite of nine applications targeted at evaluating STMs. However, the applications are implemented in Haskell and are thus not directly compatible with the majority of STM implementations or HTM and hybrid TM simulation environments. Almost all of these applications are also microbenchmarks and not full applications.

3.3 Stanford Transactional Applications for Multi-Processing (STAMP)

At present, no benchmark suite for TM covers a wide enough range of algorithms and application domains, stresses most cases of transactional behavior, and runs easily on many classes of TM systems. The *Stanford Transactional Applications for*

Multi-Processing (STAMP) is the first benchmark suite to satisfy all of these requirements [16]. STAMP consists of eight applications with 30 different sets of configurations and input data that exercise a wide range of transactional behaviors. Moreover, STAMP can run on a variety of hardware, software, and hybrid TM systems. STAMP is publicly available at `http://stamp.stanford.edu`.

3.3.1 Design Philosophy

The design of the STAMP benchmark suite follows three guiding principles to make it an effective and comprehensive tool for evaluating TM systems:

1. **Breadth:** STAMP consists of a variety of algorithms and application domains. In particular, I favor applications that are not trivially parallelizable without synchronization, as they can benefit significantly (in terms of ease of programming) from the optimistic concurrency that TM offers.

2. **Depth:** STAMP covers a wide range of transactional behaviors such as varying degrees of contention, short and long transactions, and different sizes of read and write sets. I also include applications that make frequent use of coarse-grain transactions and spend a significant portion of their execution time within transactions. This results in a balanced set of workloads that adequately stress the underlying TM system.

3. **Portability:** STAMP can easily run on many classes of TM systems, including hardware-based (HTM), software-based (STM) and hybrid designs. The code for all benchmarks is written in C with annotations to indicate both transaction boundaries and memory accesses that require instrumentation for software and hybrid systems. I have successfully run the suite on six TM systems, including HTM, STM, and hybrid TM systems with eager and lazy data versioning.

Table 3.1 summarizes how various parallel and TM benchmarks meet the three requirements discussed above. To the best of my knowledge, STAMP is the only suite that satisfies all three requirements at this point. All the parallel benchmarks (the

Table 3.1: Summary of publicly available benchmark suites used to evaluate TM systems. The bottom half of the table lists benchmarks created specifically for analyzing TM systems. The *Breadth* column also lists the number of applications or kernels included in each suite.

Benchmark	Breadth	Depth	Portability	Comments
SPLASH-2 [97]	yes (12)	no	partial	Mostly regular algorithms.
NPB OpenMP [49]	yes (7)	no	partial	Mostly regular algorithms.
SPEComp [91]	yes (11)	no	partial	Mostly regular algorithms.
BioParallel [48]	partial (5)	no	partial	Domain-specific.
MineBench [66]	partial (15)	no	partial	Domain-specific.
PARSEC [8]	yes (12)	no	partial	No transaction annotations.
RSTMv3 [59, 84]	no (6)	yes	yes	Mostly microbenchmarks.
STMbench7 [39]	no (1)	yes	yes	Single application.
Perfumo et al. [68]	yes (9)	yes	no	Mostly microbenchmarks. Implemented in Haskell.
STAMP [16]	**yes** (8)	**yes**	**yes**	**Meets all 3 requirements.**

upper half of Table 3.1) are only partially portable with regard to TM systems as they lack annotations for generating STM read and write barriers. As for the TM benchmarks (the lower half of the table), RSTMv3 [59, 84] has six benchmarks, five of which are microbenchmarks, and thus it does not satisfy the breadth requirement.

3.3.2 Applications Overview

STAMP currently consists of eight applications: `bayes`, `genome`, `intruder`, `kmeans`, `labyrinth`, `ssca2`, `vacation`, and `yada`. These applications span a variety of computing domains as well as runtime transactional characteristics such as varying transaction lengths, read and write set sizes, and amounts of contention. Table 3.2 gives a brief description of each benchmark, and more detailed descriptions follow in Section 3.4.

Table 3.2: The eight applications in the STAMP suite.

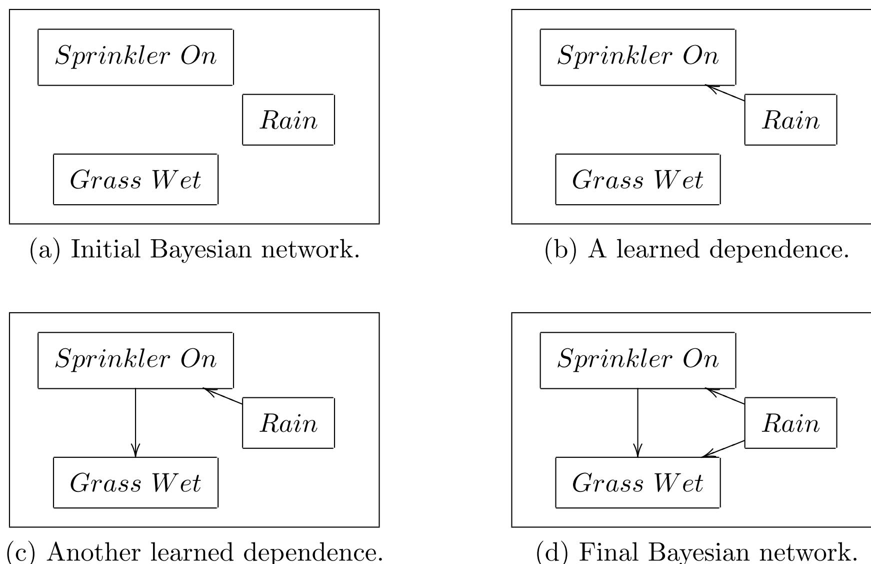
Application	**Domain**	**Description**
bayes	machine learning	Learns structure of a Bayesian network
genome	bioinformatics	Performs gene sequencing
intruder	security	Detects network intrusions
kmeans	data mining	Implements K-means clustering
labyrinth	engineering	Routes paths in maze
ssca2	scientific	Creates efficient graph representation
vacation	online transaction processing	Emulates travel reservation system
yada	scientific	Refines a Delaunay mesh

3.3.3 Implementation

To ease portability of STAMP to several TM systems, I chose to implement all the applications using the C programming language. I also used a low-level API to manually identify parallel threads and insert transaction markers and barriers. The same annotations are used by all TM versions of the code. Moreover, the only difference between the eager and lazy variants of the STM and hybrid versions of the code is linkage against a different TM barrier library. Manual optimizations of read and write barriers for accesses to shared data were performed following the guidelines in [1, 44]. Currently, the TM annotations are implemented by C macros, which makes them easy to replace, remove, or port to different systems. In a later version of STAMP, a higher-level representation for the transactions will be used and a compiler will perform the optimizations. One example of such a representation is OpenTM [5].

3.4 STAMP Application Descriptions

This section explains the algorithms and data structures present in each application as well as the parallelization strategy and use of transactions in each.



(a) Initial Bayesian network.

(b) A learned dependence.

(c) Another learned dependence.

(d) Final Bayesian network.

Figure 3.1: A Bayesian network represented as a graph, where each node represents a variable and edges indicate conditional dependences. Dependences are learned and added iteratively. In this example, *Rain* affects the state of *Sprinkler*, and *Grass* becomes wet from either *Sprinkler* or *Rain*.

3.4.1 `bayes`

This application implements an algorithm for learning the structure of Bayesian networks from observed data, which is an important part of machine learning. The algorithm uses a hill-climbing strategy that combines local and global search, similar to the technique described in [26]. An adtree data structure [64] is used to achieve efficient estimates of probability distributions. The Bayesian network itself is represented as a directed acyclic graph, with a node for each variable and an edge for each conditional dependence between variables. Figure 3.1 depicts an example Bayesian network and the iterative process used to determine its structure.

Pseudocode for the main part of the algorithm in shown in Figure 3.2. Initially, the network has no dependencies among variables, and the algorithm incrementally learns dependencies by analyzing the observed data. On each iteration, each thread is given a variable to analyze (Line 6), and as more dependencies are added to the network (Line 12), connected subgraphs of dependent variables are formed.

```
1: global Queue dependencies
2: global Graph network
3: begin parallel
4:     while true do
5:         TxBegin
6:             dependency ← dependencies.pop()
7:         TxEnd
8:         if dependency is null then
9:             break
10:        end if
11:        TxBegin
12:            network.apply(dependency)
13:        TxEnd
14:        TxBegin
15:            fromVariable ← dependency.from()
16:            subGraph ← network.subGraph(fromVariable)
17:            newDependency ← network.getNewDependency(fromVariable, subGraph)
18:        TxEnd
19:        if newDependency is not null then
20:            TxBegin
21:                dependencies.push(newDependency)
22:            TxEnd
23:        end if
24:    end while
25: end parallel
```

Figure 3.2: Pseudocode for `bayes`.

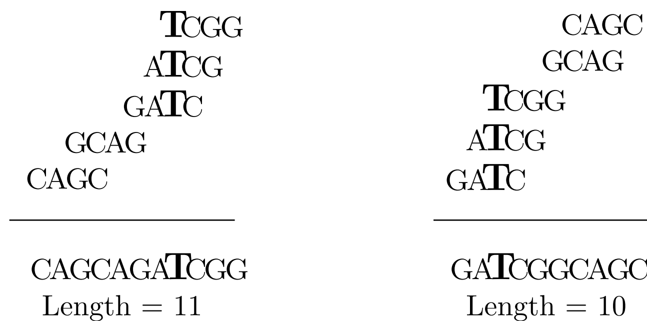


Figure 3.3: Two possible reconstructions of the segments: `ATCG`, `CAGC`, `CGAG`, `TCGG`, and `GATC`. Reconstruction is performed by sliding segments so they overlap with each other. The bottom half of the figure shows the final reconstructed genome. For example, note how all the occurrences of **T** lie in the same column. Between the two reconstructions, the one on the right is more optimal as it is shorter.

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. Using transactions is much simpler than using a lock-based approach as locks would require manually orchestrating a two-phase locking scheme with deadlock detection and recovery to allow concurrent modifications of the graph. This usage of transactions is similar to the transaction-based soft optimization techniques covered in [6]. Calculations of new dependencies (Figure 3.2, Line 17) take up most of the execution time, causing `bayes` to spend almost all its execution time in long transactions that have large read and write sets. Overall, this benchmark has a high amount of contention as the subgraphs change frequently.

3.4.2 `genome`

Genome assembly is the process of taking a large number of DNA segments and matching them to reconstruct the original source genome. An example of shotgun genome sequencing is shown in Figure 3.3.

As shown in Figure 3.4, this program has two phases to accomplish this task. Since there is a relatively large number of DNA segments, there are often many duplicates. The first phase of the algorithm utilizes a hash set to create a set of unique segments.

```
 1: global Array segments
 2: global Set uniqueSegments
 3: begin parallel
 4:     for all segment in segments.myPartition() do                     ▷ Start Phase 1
 5:         TxBegin
 6:             if segment not in uniqueSegments then
 7:                 uniqueSegments.insert(segment)
 8:             end if
 9:         TxEnd
10:     end for
11:     Barrier
12:     for all uniqueSegment in uniqueSegments.myPartition() do         ▷ Start Phase 2
13:         TxBegin
14:             matchedSegment ← uniqueSegments.findMatch(uniqueSegment)
15:             if matchedSegment is not null then
16:                 uniqueSegments.remove(matchedSegment)
17:                 uniqueSegment.join(matchedSegment)
18:             end if
19:         TxEnd
20:     end for
21: end parallel
```

Figure 3.4: Pseudocode for `genome`.

In the second phase of the algorithm, each thread tries to remove a segment from a global pool of unmatched segments and add it to its partition of currently matched segments. When performing the matching between two segments, Rabin-Karp string matching [51] is used to speed up the comparison.

Transactions are used in each of the two phases of the benchmark. Additions to the set of unique segments are enclosed by transactions to allow for concurrent accesses, and accesses to the global pool of unmatched segments are also enclosed by transactions since multiple threads may try to remove the same segment. By using transactions for the reconstruction, I did not have to implement a deadlock avoidance scheme. Overall, the transactions in `genome` are of moderate length and have moderate read and write set sizes. Additionally, almost all of the execution time is transactional, and there is relatively little contention.

3.4.3 `intruder`

Signature-based network intrusion detection systems (NIDS) scan network packets for matches against a known set of intrusion signatures. This benchmark emulates Design 5 of the NIDS described by Haagdorens et al. in [40]. Network packets are processed in parallel and go through three phases: *capture*, *reassembly*, and *detection*. The main data structure in the *capture* phase is a simple FIFO queue, and the *reassembly* phase uses a dictionary (implemented by a self-balancing tree) that contains lists of packets that belong to the same session. When evaluating their five designs for a multithreaded NIDS, Haagdorens et al. state that the complexity of the *reassembly* phase caused them to use coarse-grain synchronization in Designs 4 and 5. Thus, even though these two designs attempted to exploit higher levels of concurrency, their coarse-grain synchronization resulted in worse performance.

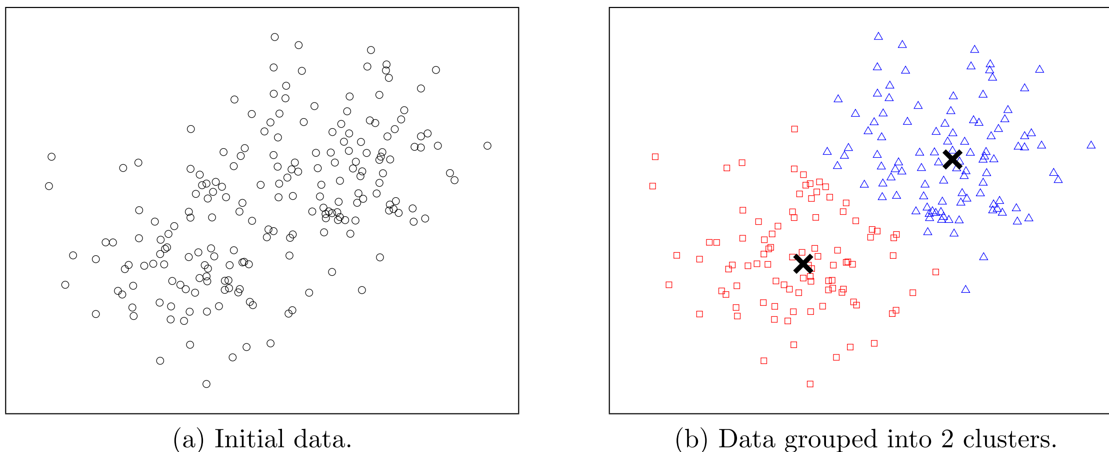
In the TM version included in STAMP, the *capture* and *reassembly* phases are each enclosed by transactions. Hence, the code for each phase is as simple as that with coarse-grain locks but hopefully achieves good performance through optimistic concurrency. When operating on these data structures, this benchmark has relatively short transactions. It also has moderate to high levels of contention depending on how often the *reassembly* phase rebalances its tree. Overall, since two of the three phases are spent in transactions, this benchmark has a moderate amount of total transactional execution time. The pseudocode for the benchmark is shown in Figure 3.5.

3.4.4 `kmeans`

The K-means algorithm groups objects in an N-dimensional space into K clusters as shown in Figure 3.6. This algorithm is commonly used to partition data items into related subsets. The implementation used is taken from MineBench [66] and has each thread processing a partition of the objects iteratively. The TM version add 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, with larger values of K resulting in less frequent conflicts as it is less likely that two threads

```
 1: global Queue packets
 2: global Decoder decoder
 3: begin parallel
 4:     Detector detector
 5:     while true do
 6:         TxBegin                                          ▷ Start Capture phase
 7:             packet ← packets.remove()
 8:         TxEnd
 9:         if packet is null then
10:             break
11:         end if
12:         TxBegin                                          ▷ Start Reassembly phase
13:             error ← decoder.reassemble(packet)
14:         TxEnd
15:         if error then
16:             Invalid packet
17:         end if
18:         TxBegin
19:             data ← decoder.getComplete(packet)
20:         TxEnd
21:         if data is not null then                         ▷ Start Detection phase
22:             error ← detector.check(data)
23:             if error then
24:                 Intrusion detected
25:             end if
26:         end if
27:     end while
28: end parallel
```

Figure 3.5: Pseudocode for `intruder`



(a) Initial data. (b) Data grouped into 2 clusters.

Figure 3.6: An example of K-means clustering with $K = 2$. In (b), the two clusters centers are indicated by the symbol $\times$.

are concurrently operating on the same cluster center. Since threads only occasionally update the same center concurrently, this algorithm benefits from TM's optimistic concurrency.

When updating the cluster centers, the size of the transaction is proportional to D, the dimensionality of the space. Thus, the sizes of the transactions in `kmeans` are relatively small and so are its read and write sets. Overall, the majority of execution time for `kmeans` is spent calculating the new cluster centers. During this operation, each thread reads from its partition of objects so no transaction is required. Thus, relatively little of the total execution time is spent in transactions.

3.4.5 `labyrinth`

Lee's algorithm [57] is a popular maze routing algorithm used in applications such as VLSI layout. To route a path, the algorithm first performs a breadth-first search from the start point to the end point, in what is called the *Expand* phase. After the end point is reached, the algorithm executes the *Backtrack* phase, which simply involves picking the final route from one of the valid paths traversed during the *Expand* phase. Figure 3.8 illustrates how these two phases work.

```
 1: global Array points                              ▷ N points of dimension D
 2: global Array memberships                         ▷ N integers ∈ [1...K]
 3: global Array centers                             ▷ K points of dimension D
 4: global Integer delta
 5: begin parallel
 6:     for all point in points.myPartition() do
 7:         k = FindNearestCenter(centers, point)
 8:         if k ≠ memberships[point] then
 9:             myDelta ← myDelta + 1
10:             memberships[point] ← k
11:         end if
12:         TxBegin
13:             UpdateCenter(centers[k], point)
14:         TxEnd
15:     end for
16:     TxBegin
17:         delta ← delta + myDelta
18:     TxEnd
19: end parallel
```

Figure 3.7: Pseudocode for `kmeans`

The `labyrinth` benchmark implements a variant of Lee's algorithm similar to the *LEE-TM-p-ws* program from [96]. The main data structure is a three-dimensional uniform grid that represents the maze. In the parallel version, each thread grabs a start and end point that it must connect by a series of adjacent maze grid points. Pseudocode for the algorithm appears in Figure 3.9.

The calculation of the path and its addition to the global maze grid are enclosed by a single transaction. A conflict occurs when two threads pick paths that overlap. To reduce the chance of conflicts, the privatization technique described in [96] is used. Specifically, a per-thread copy of the grid is created and used for the route calculation. Finally, when a thread wants to add a path to the global grid, it revalidates its work by re-reading all the grid points along the new path. If this validation fails, the transaction aborts and the process is repeated, starting with a new, updated copy of the global grid. Transactions are beneficial for implementing this program as complex deadlock detection and recovery techniques would be required in a lock-based approach.

		S			
				E	

(a) Initial maze.

		1			
	1	S	1		
		1			
				E	

(b) Starting *Expand*.

3	2	1	2	3	4
2	1	S	1	2	3
3	2	1	2	3	4
4	3	2	3	E	
	4	3	4		
		4			

(c) Finished *Expand*.

3	2	1	2	3	4
2	1	S	1	2	3
3	2	1	2	↑	4
4	3	2	3	E	
	4	3	4		
		4			

(d) Starting *Backtrack*.

3	2	1	2	3	4
2	1	S	←	←	3
3	2	1	2	↑	4
4	3	2	3	E	
	4	3	4		
		4			

(e) Finished *Backtrack*.

		•	•	•	
				•	
				•	

(f) Routed path.

Figure 3.8: Illustration of Lee's maze routing algorithm. The *Start* and *End* point are indicated by the letters *S* and *E*, respectively. During *Expand*, a breadth-first search is performed to mark the maze grid points with their distance from *S*. During *Backtrack*, an arbitrary valid path leading from the destination to the source is chosen.

```
 1: global Queue tasks
 2: global Grid maze
 3: begin parallel
 4:     while true do
 5:         TxBegin
 6:             task ← tasks.remove()
 7:         TxEnd
 8:         if task is null then
 9:             break
10:         end if
11:         start ← task.start()
12:         end ← task.end()
13:         TxBegin
14:             mazeCopy ← maze.copy()
15:             TxEarlyRelease(maze)                    ▷ Read maze while creating copy
16:             success ← Expand(mazeCopy, start, end)
17:             if success then
18:                 path ← Backtrack(mazeCopy, end, start)
19:                 valid ← maze.addPath(path)      ▷ Another thread's path could invalidate ours
20:                 if not valid then
21:                     TxRestart                               ▷ Rollback to TxBegin
22:                 end if
23:             end if
24:         TxEnd
25:     end while
26: end parallel
```

Figure 3.9: Pseudocode for `labyrinth`.

Additional performance can be achieved in the program by using early-release [46], a method that is also described in [96]. Early-release allows a transaction to remove a data address from its transactional read set so that it does not generate conflicts. However, the programmer or compiler must guarantee that removing the address from the read set does not violate the atomicity or consistency of the program. In `labyrinth`, early-release is used to remove all transactional reads generated when copying the global grid to the per-thread copy (Figure 3.9, Line 15). As early-release operates with cache line granularity, each maze grid point is padded to occupy an entire cache line to ensure correctness. Finally, since early-release is not a feature available on all TM systems, its use can be disabled when compiling this benchmark.

Overall, almost all of `labyrinth`'s execution time is taken by the path calculation, and this operation also reads and writes an amount of data proportional to the number of total maze grid points. Consequently, `labyrinth` has very long transactions with very large read and write sets. Virtually all of the code is executed transactionally, and the amount of contention is very high because of the large number of transactional accesses to memory.

3.4.6 `ssca2`

The Scalable Synthetic Compact Applications 2 (SSCA2) benchmark [4] is comprised of four kernels that operate on a large, directed, weighted multi-graph. These four graph kernels are commonly used in applications ranging from computational biology to security. For STAMP, I focus on Kernel 1, which constructs an efficient graph data structure using adjacency arrays and auxiliary arrays. This part of the code is well suited for TM as it benefits greatly from optimistic concurrency.

The transactional version of SSCA2 has threads adding nodes to the graph in parallel and uses transactions to protect accesses to the adjacency arrays as shown in Figure 3.10. Since this operation is relatively small, not much time is spent in transactions. Additionally, the length of the transactions and the sizes of their read and write sets is relatively small. The amount of contention is the application is

```
1:  global Array nodes
2:  global Array edges
3:  begin parallel
4:      for all edge in edges.myPartition() do
5:          fromIndex ← edge.from()
6:          toIndex ← edge.to()
7:          TxBegin
8:              node[fromIndex].addChild(toIndex)
9:          TxEnd
10:     end for
11: end parallel
```

Figure 3.10: Pseudocode for `ssca2`.

also relatively low as the large number of graph nodes leads to infrequent concurrent updates of the same adjacency list.

3.4.7 `vacation`

This application implements an online transaction processing system similar in design to SPECjbb2000 [90] but serving the task of emulating a travel reservation system instead of a wholesale company. Figure 3.11 shows the three-tier organization of `vacation`. The system is implemented as a set of trees that keep track of customers and their reservations for various travel items. During the execution of the workload, several client threads perform a number of sessions that interact with the travel system's database. In particular, there are three distinct types of sessions: reservations, cancellations, and updates. Pseudocode for the application is shown in Figure 3.12, and the usage of transactions is based on the approached described in [28].

Each of these client sessions is enclosed in a coarse-grain transaction to ensure validity of the database. Consequently, `vacation` spends a lot of time in transactions and its transactions are of medium length with moderate read and write set sizes. Low to moderate levels of contention among threads can be created by increasing the fraction of sessions that modify large portions of the database. Finally, using transactions greatly simplified the parallel programming process as designing an efficient locking strategy for all the data structures in `vacation` is non-trivial.

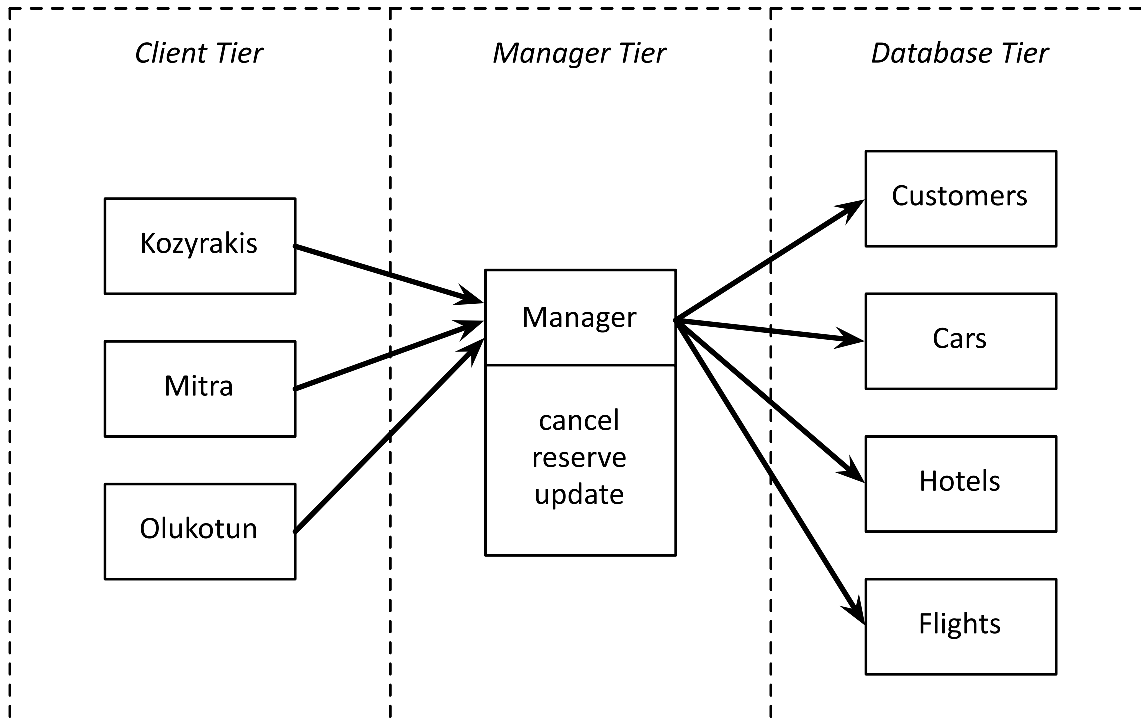


Figure 3.11: The three-tier design of `vacation` is similar to that found in SPECjbb2000 [90]. Clients can *reserve* items, *cancel* reservations, or *update* existing reservations. These task requests are processed by the travel reservation system's manager, which updates the database information on *customers*, *cars*, *hotels*, and *flights*.

```
 1: global Manager manager
 2: begin parallel
 3:     for i ∈ [1...N] do
 4:         action ← RandomAction()
 5:         items ← RandomItems()
 6:         TxBegin
 7:             if action is reserve then
 8:                 manager.reserve(items)
 9:             else if action is cancel then
10:                 manager.cancel(items)
11:             else if action is update then
12:                 manager.update(items)
13:             end if
14:         TxEnd
15:     end for
16: end parallel
```

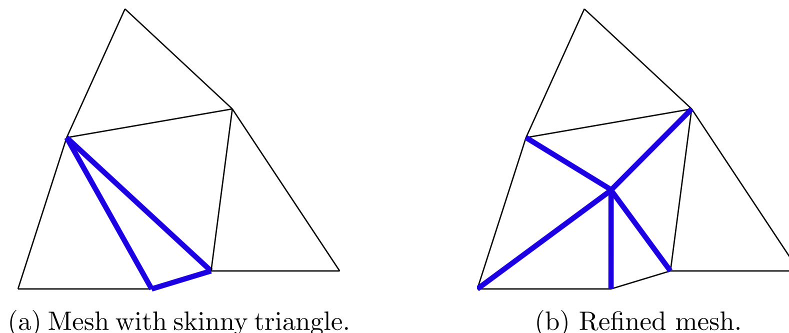
Figure 3.12: Pseudocode for `vacation`

3.4.8 `yada`

Meshes of triangles or tetrahedra are commonly used by numerical methods to solve problems in applications such as graphics rendering or partial differential equation solvers. A popular method for producing unstructured meshes is *Delaunay triangulation*, which guarantees meshes with a minimum desired angle while simultaneously keeping the total number of triangles relatively small. These two properties lead to the good performance of Delaunay meshes in numerical methods.

Delaunay mesh refinement is a technique for generating Delaunay meshes. It begins with an initial Delaunay triangulation and then iteratively retriangulates regions of the mesh until the desired minimum angle is reached. Figure 3.13 illustrates the process of performing Delaunay triangulation on a mesh region that contains a triangle that is too skinny (as one of its angle is too small).

The `yada` (Yet Another Delaunay Application) benchmark implements Ruppert's algorithm for Delaunay mesh refinement [78]. The basic data structures are a graph that stores all the mesh triangles, a set that contains the mesh boundary segments, and a task queue that holds the triangles that need to be refined. In each iteration



(a) Mesh with skinny triangle. (b) Refined mesh.

Figure 3.13: An example of Delaunay mesh refinement. On the left, the skinny triangle is indicated by bold line segments. On the right, the bold line segments represent the retriangulation.

of the algorithm, a skinny triangle is removed from the work queue, its retriangulation is performed on the mesh, and any new skinny triangles that result from the retriangulation are added to the work queue. If a retriangulation involves a mesh boundary segment, the mesh boundary segment is first bisected to ensure that the retriangulation does not generate elements outside of the mesh boundary.

The usage of transactions in `yada` is shown in Figure 3.14. The transactional structure is similar to that in [54], but it is applied to a different algorithm in this benchmark. Accesses to the work queue are enclosed by a transaction as is the entire refinement of a skinny triangle. As almost all the execution time is spent calculating the retriangulation of a skinny triangle, this benchmark has relatively long transactions and spends almost all of its execution time in transactions. While performing the retriangulation, several triangles in the mesh are visited and later modified, leading to large read and write sets and a moderate amount of contention. As `yada` continuously concurrently modifies a shared graph, it was simpler for me to parallelize the algorithm with TM than with locks.

3.5 Configurations and Data Sets

One goal for STAMP is to cover a wide range of transactional execution behaviors in order to stress all aspects of the evaluated TM systems. The differences across

```
 1: global Mesh mesh
 2: global Set segments
 3: global PriorityQueue work
 4: begin parallel
 5:     while true do
 6:         TxBegin
 7:             element ← work.remove()
 8:         TxEnd
 9:         if element is null then
10:             break
11:         end if
12:         point ← element.center()
13:         TxBegin
14:             if element is segment then
15:                 segments.split(element)
16:             end if
17:             region ← mesh.affectedRegion(point)
18:             newRegion ← region.retriangulate(point)
19:             mesh.replaceRegion(region, newRegion)
20:         TxEnd
21:         TxBegin
22:             work.insert(newRegion.badTriangles())
23:         TxEnd
24:     end while
25: end parallel
```

Figure 3.14: Pseudocode for `yada`.

Table 3.3: Qualitative summary of each STAMP application's runtime transactional characteristics: length of transactions (number of instructions), size of the read and write sets, time spent in transactions, and amount of contention. The description of each characteristic is relative to the other STAMP applications. A quantitative characterization appears later in Section 4.2.

Application	**Tx Length**	**R/W Set**	**Tx Time**	**Contention**
bayes	Long	Large	High	High
genome	Medium	Medium	High	Low
intruder	Short	Medium	Medium	High
kmeans	Short	Small	Low	Low
labyrinth	Long	Large	High	High
ssca2	Short	Small	Low	Low
vacation	Medium	Medium	High	Low/Medium
yada	Long	Large	High	Medium

the applications discussed above achieve this to some extent: the applications exhibit different transaction lengths, read and write set sizes, percentage of time spent in transactions, amount of contention, and working set size. The transactional characteristics of each of the STAMP applications are summarized qualitatively in Table 3.3.

To provide further coverage, I also exploit the fact that several of the applications exhibit different behavior depending on the size and type of the input data set and the value of certain configuration parameters. Table 3.4 lists each of the STAMP applications and their recommended configurations and data sets. There are six variants of `kmeans` and `vacation` to target different levels of contention and working set sizes. These variants are denoted by appending *-low* and *-high* to the application name to indicate the relative amount of contention. Additionally, the usage of a larger data set is indicated by adding a '+' to the end of the application name. Thus, the six variants of `kmeans` are named `kmeans-high`, `kmeans-high+`, `kmeans-high++`, `kmeans-low`, `kmeans-low+`, and `kmeans-low++`. For the remainder of the benchmarks, there are only three variants as increasing the data set size also

Table 3.4: Recommended configurations and data sets for STAMP. Suffixes of *-low* and *-high* indicate the relative amount of contention, and appended '+' symbols indicate larger input sizes.

Application	Arguments	Description
bayes	-v32 -r1024 -n2 -p20 -i2 -e2	Dependencies for v variables are learned from r records, which have $n \times p$ parents per variable on average. Edge insertion has a penalty of i, and up to e edges are learned per variable.
bayes+	-v32 -r4096 -n2 -p20 -i2 -e2	
bayes++	-v32 -r4096 -n10 -p40 -i2 -e8 -s1	
genome	-g256 -s16 -n16384	Gene segments of s nucleotides are sampled from a gene with g nucleotides. A total of n segments are analyzed to reconstruct the original gene.
genome+	-g512 -s32 -n32768	
genome++	-g16384 -s64 -n16777216	
intruder	-a10 -l4 -n2048 -s1	n traffic flows are analyzed, a of which have attacks injected. Each flow has a max of l packets, and the random seed s is used.
intruder+	-a10 -l16 -n4096 -s1	
intruder++	-a10 -l128 -n262144 -s1	
kmeans-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16	The number of cluster centers used is varied from m to n. A convergence threshold of t is used, and analysis is performed on input i. The input consists of n points of d dimensions generated about c centers.
kmeans-high+	-m15 -n15 -t0.05 -i random-n16384-d24-c16	
kmeans-high++	-m15 -n15 -t0.00001 -i random-n65536-d32-c16	
kmeans-low	-m40 -n40 -t0.05 -i random-n2048-d16-c16	
kmeans-low+	-m40 -n40 -t0.05 -i random-n16384-d24-c16	
kmeans-low++	-m40 -n40 -t0.00001 -i random-n65536-d32-c16	
labyrinth	-i random-x32-y32-z3-n96	The input i consists of a maze of dimensions $x \times y \times z$. n paths are routed.
labyrinth+	-i random-x48-y48-z3-n64	
labyrinth++	-i random-x512-y512-z7-n512	
ssca2	-s13 -i1.0 -u1.0 -l3 -p3	There are 2^s nodes in the graph. The probability of inter-clique edges and unidirectional edges are i and u, respectively. The max path length is l, and the max number of parallel edges is p.
ssca2+	-s14 -i1.0 -u1.0 -l9 -p9	
ssca2++	-s20 -i1.0 -u1.0 -l3 -p3	
vacation-high	-n4 -q60 -u90 -r16384 -t4096	The database has r records of each reservation item, and clients perform t sessions. Of these sessions, u% reserve or cancel items and the remainder create or destroy items. Sessions operate on up to n items and are performed on q% of the total records.
vacation-high+	-n4 -q60 -u90 -r1048576 -t4096	
vacation-high++	-n4 -q60 -u90 -r1048576 -t4194304	
vacation-low	-n2 -q90 -u98 -r16384 -t4096	
vacation-low+	-n2 -q90 -u98 -r1048576 -t4096	
vacation-low++	-n2 -q90 -u98 -r1048576 -t4194304	
yada	-a20 -i 633.2	The input mesh i is refined so that it has a minimum angle of a. The input *633.2* consists of 1264 elements; *ttimeu10000.2*, 19998 elements; and *ttimeu1000000.2*, 1999998 elements.
yada+	-a10 -i ttimeu10000.2	
yada++	-a15 -i ttimeu1000000.2	

affects the level of contention. For example, `bayes` has the variants: `bayes`, `bayes+`, and `bayes++`. Finally, the variants without a '++' suffix are intended for running in simulation environments.

Chapter 4

Characterizing and Applying the Stanford Transactional Applications for Multi-Processing

Most events or actions that seem strange actually have logical explanations. Oftentimes, the behaviors of Transactional Memory systems are quite puzzling; however, a well-designed set of benchmarks can help shed insights on performance bottlenecks and motivate design improvements. In this chapter, I quantitatively characterize the runtime transactional characteristics of STAMP to show how it meets the requirements of a well-designed TM benchmark suite and then demonstrate how STAMP can be a useful tool for evaluating TM systems.

4.1 Methodology

Two sets of experiments were used to evaluate STAMP. The first set quantitatively verifies the coverage of transactional behaviors by the benchmark suite. The second set demonstrates the portability and practical usefulness of STAMP by running the suite on several different TM designs and comparing their performance. To ensure a valid comparison between HTM and STM systems, I used an execution-driven simulator for all experiments. Table 4.1 presents the main parameters of the simulated

Table 4.1: Configuration for the simulated multi-core system.

Feature	Description
Processors	1 to 32 x86 cores, in-order, single-issue
L1 Cache	64KB, private, 4-way associativity, 32B line, 1-cycle access Provides TM bookkeeping for HTM systems
Network	32B bus, split transactions, pipelined, MESI
L2 cache	8MB, shared, 32-way associativity, 32B line, 12-cycle access
Memory	100-cycle off-chip access

multi-core system I used. The processor model assumes an IPC of 1 for all instructions that do not access memory. However, the simulator captures all the memory hierarchy timings including contention and queuing events.

Using the simulator, STAMP was run on the four following TM systems:

- **Lazy HTM:** This is the HTM system described in Section 2.4.1. It follows the TCC architecture [41], except that transactions are only executed for code sections demarcated by transaction boundary markers instead of all code sections. It implements lazy data versioning in caches and performs conflict detection late (when a transaction is ready to commit) by using the coherence protocol. Because the coherence protocol is used for conflict detection, the lazy HTM detects conflicts at cache-line granularity. When the lazy HTM overflows the caches' capacity for buffering speculative data, it temporarily serializes the execution of transactions. On conflicts, the lazy HTM restarts the aborted transaction immediately, without using any backoff schemes. Note that there is no fundamental reason why HTM cannot implement more sophisticated contention management; not using backoff was just a simple HTM design point I used.

- **Eager HTM:** The HTM system described in Section 2.4.2. It is similar to LogTM [65] and uses the L1 caches and the coherence protocol to implement eager data versioning and early conflict detection, respectively. On conflicts, the requester loses, aborts, and restarts immediately without any backoff. Conflicts

are detected at line granularity, and to prevent livelock, transactions are given high priority after aborting 32 times. There is no fundamental reason why eager HTM cannot use more sophisticated contention management (e.g., temporarily stalling transactions), and the previously described policy is simply one I chose. Finally, overflowed addresses are handled by inserting them into a Bloom filter [9]. The Bloom filter participates in the conflict detection mechanism and because of its conservative nature (aliasing of addresses), it may signal false conflicts.

- **Lazy STM:** An x86 port of the TL2 STM system [32] that is described in Section 2.5.1. It performs lazy versioning using a software write buffer. To provide conflict detection, it uses locks for data in the write set during commit. Conflicts for data in the read set are detected by checking version numbers periodically, and after a transaction aborts three times, the lazy STM uses a randomized linear backoff mechanism. Finally, it detects conflicts at word granularity and provides weak isolation of transactions.

- **Eager STM:** An eager version of TL2 as described in Section 2.5.2. It uses an undo log and holds locks on data in the write set throughout the transaction to provide versioning. Conflict detection is similar to that of the lazy STM system, and its conflict management, conflict detection granularity, and weak isolation policies are the same as in the lazy variant.

I selected these TM systems as representative points in the HTM and STM design space. The conventional wisdom is that HTM systems should perform the best as hardware transparently performs all transactional bookkeeping. A previous study has shown that HTM systems are up to a factor of four times faster than STM [17]. Similarly, it is commonly thought that eager systems should perform better than lazy systems as eager systems perform their memory updates throughout the transaction, instead of waiting till the transaction commit point [65, 80].

STAMP was used to evaluate if the conventional wisdom holds for these four TM designs. A good benchmark suite that stresses a wide range of potential transactional behaviors may be able to identify unexpected cases in which the conventional wisdom

is wrong. I used the same code (same parallelization strategy and same transaction boundaries) with all systems. The STM systems used the optimized annotations for read and write barriers (explicit barriers), but each of them provided different code for the actual barrier functionality. When compiling for HTM systems, the annotations for read and write barriers were ignored (implicit barriers). For the application configurations and data sets, those without the '++' suffix were used as those are the ones designed for use with a simulation environment.

4.2 Basic Characterization

In this section, I present a quantitative analysis of the transactional characteristics of STAMP. Table 4.2 presents the basic runtime statistics for the STAMP applications and includes data such as the transaction length in instructions, read and write set size in 32-byte cache lines, percentage of execution time spent in transactions, and average number of retries per transaction. All of these numbers were measured using the lazy HTM system in order to shield these statistics from the implementation details of the barriers necessary for memory accesses to shared variables in STM systems. For the number of read and write barriers and number of STM retries, however, the STM was used.

All measured transactional characteristics in Table 4.2 vary at least two orders of magnitude, thus showing that STAMP is able to cover many different transactional execution scenarios. In most of the applications, the transactional statistics follow a normal distribution, but for `genome` and `intruder` they are bimodal and for `vacation` they are trimodal. The modes that arise in these applications are caused by the phased nature of their execution.

4.2.1 Transaction Length

The mean number of instructions per transaction ranges from a low of 50 instructions in `bayes` to a high of 687,809 in `labyrinth+`. In `labyrinth`, the large set of operations

Table 4.2: The basic characterization of the STAMP applications. The number of instructions per transaction does not include instructions that occur as part of TM barriers. The lazy HTM was used to measure the read and write sets and the amount of time spent in transactions. The amounts of read and write barriers were collected using the lazy STM, and the lazy HTM was used to find the fraction of time spent in transactions. For the number of retries, 16 threads were used on all systems. The transactional statistics for `genome` and `intruder` follow bimodal distributions, and those for `vacation` are trimodal. The rest of the applications have normal distributions.

Application	Per Transaction					Time in Transactions	Retries Per Transaction				Working Set	
							HTM		STM			
	Instructions (mean)	**Read Set** (90 pctile)	**Write Set** (90 pctile)	**Read Barrier** (90 pctile)	**Write Barrier** (90 pctile)		**Lazy** (mean)	**Eager** (mean)	**Lazy** (mean)	**Eager** (mean)	**Small** (KB)	**Large** (MB)
bayes	60,584	452	304	24	9	83%	0.66	6.50	0.59	0.66	128	2
bayes+	57,130	448	266	26	9	83%	0.69	5.78	0.61	0.69	128	2
genome	1,717	98	15	32	2	97%	0.10	0.47	0.14	2.20	128	1
genome+	1,709	108	15	30	2	97%	0.02	0.26	0.06	1.14	128	4
intruder	330	51	20	71	16	33%	1.79	6.27	3.54	3.31	32	1
intruder+	331	54	18	54	9	43%	0.67	2.05	1.95	2.96	128	2
kmeans-high	117	14	5	17	17	7%	0.07	0.13	2.73	3.10	16	1
kmeans-high+	153	16	6	25	25	6%	0.05	0.11	3.49	3.68	16	2
kmeans-low	117	14	5	17	17	3%	0.02	0.05	0.89	0.80	16	1
kmeans-low+	153	16	6	25	25	3%	0.01	0.02	0.81	0.70	16	2
labyrinth	219,571	433	458	35	36	100%	0.72	2.64	0.94	1.11	64	1
labyrinth+	687,809	783	779	46	47	100%	2.55	10.59	1.07	1.38	128	2
ssca2	50	10	4	1	2	17%	0.01	0.01	0.00	0.01	256	2
ssca2+	50	10	4	1	2	16%	0.00	0.00	0.00	0.00	512	4
vacation-high	3,223	130	24	432	12	86%	0.37	1.01	0.00	0.01	256	2
vacation-high+	4,193	173	23	608	12	92%	0.25	0.66	0.04	0.05	512	4
vacation-low	2,420	99	22	287	8	86%	0.07	0.25	0.00	0.00	256	2
vacation-low+	3,161	126	22	401	8	92%	0.05	0.18	0.02	0.03	512	4
yada	9,795	250	142	256	108	100%	0.52	3.06	2.51	4.35	32	2
yada+	11,596	274	145	282	108	100%	0.45	2.04	1.38	2.52	64	8

necessary to find a routing path are packed in a single atomic block. While this coarse-grain approach greatly simplifies code management, it leads to large transactions. In addition to `labyrinth`, `bayes` and `yada` also have relatively long transactions.

4.2.2 Read and Write Set Sizes

The sizes of transactional read and write sets were found by running STAMP on the lazy HTM. The 90th percentile of the data is given as a guide to TM system designers for sizing their hardware transactional buffers to handle the common case. The largest read and write set sizes are found in `labyrinth+`, with values of 783 and 779 cache lines, respectively (24.5 KB and 24.3 KB, respectively). This implies that for HTM systems, transactional support at the L2 cache may be necessary to avoid the overhead of virtualization mechanisms, especially because of associativity misses. At the other extreme is `ssca2` with a read set of 10 lines and a write set of 4 lines. Since most of the write sets are small, not stalling conflicting transactions in my eager HTM is acceptable since applying the undo log is not too expensive.

The numbers of read and write barriers were measured with the lazy STM and would have been identical had the eager STM been used instead. The number of read barriers varies from a low of 1 in `ssca2` to a high of 608 in `vacation-high+`. In comparison, the statistics for the write barriers range from 2 in `genome` and `ssca2` to 108 in `yada`. Overall, the number of read barriers is typically much larger than the number of write barriers, suggesting that designers of STMs should especially implement read barriers with low overhead.

4.2.3 Time in Transactions

Over 80% of the execution time is spent in transactions by five of the eight applications: `bayes`, `genome`, `labyrinth`, `vacation`, and `yada`. These applications use transactions frequently as their algorithms continuously operate on shared data structures. Such applications put great stress on the underlying TM system as any inefficiencies will be amplified by the frequent use of transactions. STAMP also covers applications that use transactions sporadically. Three of the benchmarks (`intruder`, `kmeans`, and

ssca2) spend less than half of their execution time in transactions, as there are significant portions of the code that operate on easily-identified private data. Over all the STAMP applications, the time spent in transactions ranges from a low of 3% in kmeans-low to a high of 100% in labyrinth and yada.

4.2.4 Transaction Retries

The last transactional characteristic measured in Table 4.2 is the average number of times a transaction retries before successfully committing. This statistic was measured by using 16 threads on the four TM systems, and it is highly dependent on both the number of threads and the underlying TM system. Moreover, the actual amount of work lost is a function of both the number of retries and the point in the transaction at which the retry occurs. Nevertheless, the data for the number of retries gives quantitative evidence that the STAMP applications cover a wide spectrum of contention cases ranging from virtually no retries (<0.01 for ssca2+) to 10.59 retries for labyrinth+. Finally, good contention management policies should be able to reduce the number of retries in most of the TM systems, and the numbers for this transactional characteristic show that STAMP can be used to evaluate contention management policies as well.

4.2.5 Working Set Size

The last two columns of Table 4.2, indicate two of the working sets exhibited by each of the programs. These values were found by setting the cache size to all power-of-2 sizes from 16 KB to 64 MB and looking for points at which significant changes in the miss rate occurred. Like the other columns in the table, these values cover a variety of different scenarios. Applications in which the small working set exceeds the capacity of the L1 cache (e.g., ssca2 and vacation) are likely to have a significant portion of the execution time spent on cache misses.

4.3 HTM and STM Performance Analysis

I measured the performance of 20 variants of the STAMP applications on the four TM systems as I varied the number of cores from 1 to 32. The speedup curves (normalized to sequential execution with code that does not have extra overhead from the annotations for threads, transactions, or barriers) are shown in Figure 4.1.

4.3.1 `bayes`

This application has the interesting result that the relative performance among the TM systems is the *opposite* of the expected ranking. As shown in Table 4.2, `bayes` has relatively large read and write sets. Consequently, both of the HTMs experience overflows (about 10% of the transactions) and suffer large performance hits.

In particular, the lazy HTM handles overflows by temporarily serializing execution of transactions and the eager HTM handles overflows by using a Bloom filter [9] to summarize the overflowed addresses. Because the Bloom filter is conservative in nature, it can cause the eager HTM to abort transactions more than necessary (but never less). Transaction aborts are especially expensive in eager data versioning TM systems as they have to process the undo logs to rollback a transaction.

As a result, the eager HTM performs the worst among all the TM systems and the lazy HTM does not perform much better. In contrast, because the STMs implement data versioning in software, they have almost no overflow-related performance penalties and perform much better than the HTMs.

The last thing to note about `bayes` is that the execution time is sensitive to the order in which dependencies are learned. Thus, the speedup curves are not as smooth as those for other STAMP applications.

4.3.2 `genome`

Early conflict detection turns out to be disadvantageous in `genome`. For example, the eager STM has an average number of retries per transaction almost 20× that of the

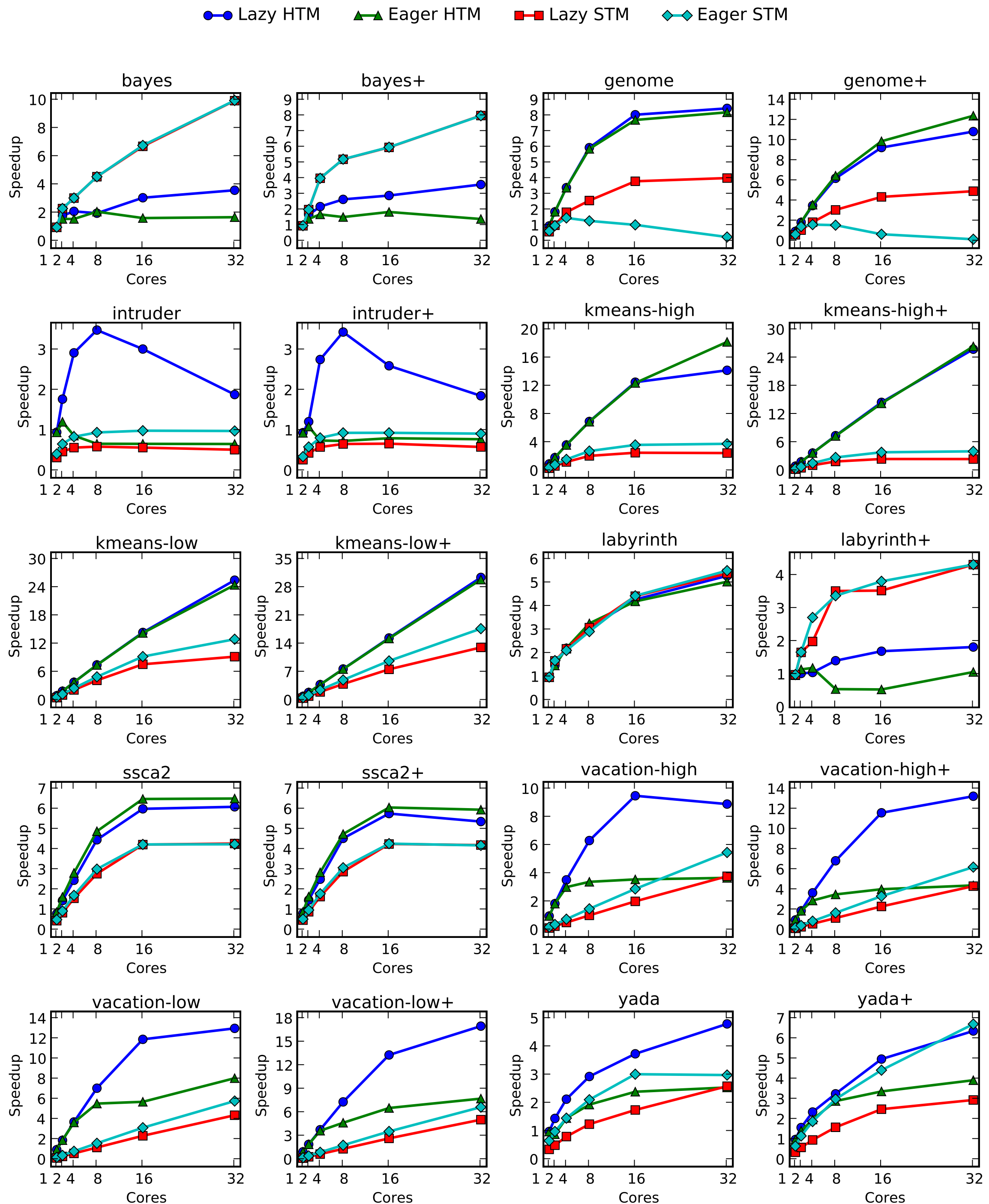


Figure 4.1: STAMP speedups over sequential code on each of the HTM and STM implementations. Application variants that use the larger data set are indicated by a '+' appended to the application name. The suffixes *-low* and *-high* indicate variants with low and with high contention, respectively.

lazy STM. This causes the eager STM to experience a high number of aborts and suffer from livelock.

In comparison, the eager HTM's contention management policy prevents it from falling prey to this pathology. In particular, when a transaction aborts several times in the eager HTM, the eager HTM system promotes the transaction's priority level and prevents any other transactions from aborting it. This technique allows the eager HTM to continue making forward progress and results in its good performance on `genome`. Overall, the HTMs are about twice as fast as the lazy STM, and the eager STM fails to scale because its eager conflict detection results in livelock. Finally, the performance of the eager STM on `genome` will likely improve with a better contention management policy that addresses livelock problems.

4.3.3 `intruder`

One advantage that the STMs have over the HTMs arises under workloads with high contention. TMs with a software component can have more sophisticated contention management policies, and in particular, the STMs use a randomized linear backoff scheme after a transaction aborts. Additionally, the overhead caused by resetting the transaction state in software (e.g., processing the software undo logs) also serves as a backoff mechanism. In contrast, the two HTM implementations do not use a backoff scheme and simply restart the transaction as soon as possible. Not having a sophisticated contention management policy is not intrinsic to HTM, and I just chose to have a simple design.

Among the STAMP applications, `intruder` has a relatively high amount of contention. Thus, whereas the performance of the STMs follow expected behavior, the HTMs do not perform as well as anticipated. The effect is more pronounced in the eager HTM, because aborts are more expensive in an eager scheme (time to apply the undo log). The early conflict detection in eager HTM also causes this program to suffer from livelock. Hence, for this application, there is an HTM system (eager HTM) that performs worse than the STMs, and neither of the HTMs scale beyond 8 cores.

Overall, the lazy HTM performs the best, and with more sophisticated contention management, the HTMs will likely scale better on this benchmark.

4.3.4 `kmeans`

In this benchmark, all four TM systems scale similarly and their relative performance follows the expected behavior; however, the actual speedups differ greatly. Because `kmeans` uses transactions infrequently and has relatively short transactions with few conflicts, the performance difference between the corresponding eager and lazy variants is very small for the HTM designs. For the STMs, however, the eager variant is noticeably faster than the lazy one because the eager STM has shorter read barriers (the read barrier in the lazy STM must first search in the write buffer to check if the data have been previously written within the same transaction). Overall, the HTMs perform 2–4× better than the STMs.

4.3.5 `labyrinth`

In this benchmark, early-release is used to remove the reads generated by the grid copying operation from the transactional read set. The usage of early-release is necessary for the HTMs as all memory accesses are transparently tracked by the hardware (i.e., implicit barriers). For the STMs, however, the TM system relies on the annotation of accesses to shared objects with read and write barriers. Thus, by not using any read barriers to perform the grid copying, the STMs avoid the need for an early-release pass over all the grid points. As a result, the number of read and write barriers is only proportional to the length of the routed path instead of to the total number of grid points. Therefore, since there are relatively few TM barriers, all the TM systems perform similarly, with the HTMs slightly worse than the STMs because of the extra early-release pass.

Performance differences among the TM systems occur in `labyrinth+` because of the larger data set. As noted in Section 4.2, the transactional read and write sets for `labyrinth+` are comparable to the size of the L1 cache. This causes both HTMs to overflow (about 30% of the transactions) and pay performance penalties. Moreover,

recall that the eager HTM handles overflows by recording overflowed addresses in a Bloom filter. To ensure correctness, the eager HTM cannot perform early-release on addresses that hit in the Bloom filter. This causes the eager HTM to perform worse than the lazy HTM as its transactions abort more frequently. In contrast to the HTMs, the STMs do not experience overflow-related performance problems as the data versioning is handled in software. Thus, their performance is about the same as with the small data set.

4.3.6 `ssca2`

Like `kmeans`, `ssca2` has very short transactions with very small read and write sets, and less than 20% of the execution time is spent in transactions. Consequently, all TM systems perform well and operate with minimal overhead. Overall, the relative ranking among the four TM systems is as expected.

4.3.7 `vacation`

All the TM systems follow the expected behavior in `vacation` except for the eager HTM. As in `bayes`, conflict detection at a finer granularity is particularly advantageous, and Table 4.2 shows that the STMs' word-level granularity conflict detection leads to much fewer aborts than the HTMs' line-level granularity conflict detection. However, for the STMs, the performance advantage gained by finer granularity conflict detection is outweighed by the overhead caused by the large number of read barriers used to provide concurrent accesses to the database trees.

Finally, for the HTMs, lazy data versioning is more advantageous than eager data versioning. In the latter scheme, transaction aborts are more expensive as an eager data versioning TM system must apply undo logs to rollback a transaction. Thus, when coupled with the moderately long transactions in `vacation`, the eager HTM performs much worse than the lazy HTM. At 32 cores in `vacation-high`, the large amount of work lost to violations in the eager HTM even causes its performance to fall below that of the STMs.

4.3.8 yada

In yada, transactions have relatively large read and write sets, which cause the HTMs to suffer from overflows. In the lazy HTM, overflows cause temporary serialization of transactions, and in the eager HTM, overflows cause more frequent transaction aborts. On the other hand, the large numbers of read and write barriers do not cause the STMs to incur a significant performance penalty.

As a result, in yada the lazy HTM performs the best and the eager HTM's speedup is about the same as the STMs. With the larger data set in yada+, however, overflows are more expensive because transactions are longer. This causes the lazy HTM to serialize transactions for longer periods of time and the eager HTM to lose more work to transaction aborts. Because of this, the eager STM is able to outperform both HTMs. The lazy STM, however, still has the worst performance because the application's large read set causes the read and commit barriers to have significant overhead.

4.4 HTM and STM Performance Summary

In general, the applications in STAMP performed well with all four TM systems. In most cases, there were significant speedups and the relative differences among HTM and STM systems were as expected. However, there were a few cases where conventional wisdom did not hold. In bayes and labyrinth+, for example, the performance of the HTMs suffered because the large number of transactional reads generated overflows, causing either transaction execution to become serialized (lazy HTM) or excessive aborts to occur (eager HTM). In bayes, the finer granularity in conflict detection allowed STMs to unexpectedly outperform the HTMs. Comparing the HTM systems, the results indicate that the two schemes (eager and lazy) either perform similarly (e.g., genome and kmeans), or the lazy HTM system performed better as it guarantees forward progress in high contention scenarios (e.g., intruder and vacation-high).

Finally, none of the shortcomings identified above in the four TM systems are necessarily fundamental. Future research may be able to address them and remove the

resulting performance loss. The key point is that STAMP's coverage of a diverse set of scenarios in transactional execution allows us to stress each TM system thoroughly and identify its (sometimes unexpected) shortcomings. Hence, STAMP can be a valuable tool for future research on TM systems.

Chapter 5

Signature-accelerated Transactional Memory

All Transactional Memory systems implement two basic mechanisms: data versioning and conflict detection. HTM systems use hardware caches for data versioning and leverage cache coherence protocols to provide conflict detection among concurrent transactions [41, 65]. Since the processor can transparently track loads and stores for the transactional bookkeeping, HTMs have low transactional overhead; however, the complexity and cost of redesigning the caches and coherence protocols to support TM can be significant. In contrast, STM adds instrumentation code (read and write barriers) to interact with an STM software library [32, 43, 80]. Since STMs are software-based, they are more cost-effective and flexible than HTMs; however, the software barriers and libraries incur much greater overhead.

Another important difference between HTM and STM is in their semantics. *Strong isolation* is the property where a TM system guarantees that code in transactions runs isolated from both transactional and non-transactional code [56]. HTMs naturally have strong isolation as the processor sees all memory accesses. On the other hand, STMs must add extra instrumentation code on non-transactional loads and stores to provide strong isolation. Since this is an additional source of overhead, high-performance STMs often sacrifice strong isolation. As a result, these STM systems may produce incorrect or unpredictable results even for simple parallel programs that

would work correctly with lock-based synchronization [34, 56, 86]. For STMs that do provide strong isolation, however, it is possible for compilers to reduce some of this overhead [86], but these techniques are not applicable to all languages and runtimes.

Given the desirable traits of TM and the tradeoffs between HTM and STM, an ideal system would be a hybrid design that combines the performance and semantics of HTM with the flexibility and cost-effectiveness of STM. *Signature-accelerated Transactional Memory (SigTM)* is a new hybrid TM design that achieves this by using small hardware signatures to accelerate software transactions and transparently provide strong isolation.

5.1 HTM and STM Overhead

STM transactions run slower than HTM transactions due to the overhead of software-based versioning and conflict detection. Even though the two systems may allow the same degree of concurrency and exhibit similar scaling, the latency of individual transactions is a key parameter for the overall performance. Figure 5.1 shows the execution time breakdown for the lazy STM running on a single processor (no conflicts or aborts), and Figure 5.2 displays the corresponding data for the eager STM. Execution time is broken into "busy" (useful instructions and cache misses) and time spent in the STM to perform transaction commit, write barriers, read barriers, and other miscellaneous functions (e.g., starting a transaction). The execution time shown is normalized to that of the sequential code.

On average, the lazy and eager STMs are slower than sequential code by 2.95× and 2.27×, respectively, on STAMP. However, in applications like `vacation`, the STM overhead can be very significant and cause slowdowns of up to 7.4×. In contrast, the overhead of HTM execution on one processor is less than 5% for both HTM systems on all of the STAMP applications. The only two applications that do not suffer significant STM overhead are `bayes` and `labyrinth`. As Table 4.2 shows, these two applications touch a relatively small amount of data for the amounts of work done in their transactions and are thus able to amortize the cost of the TM barriers.

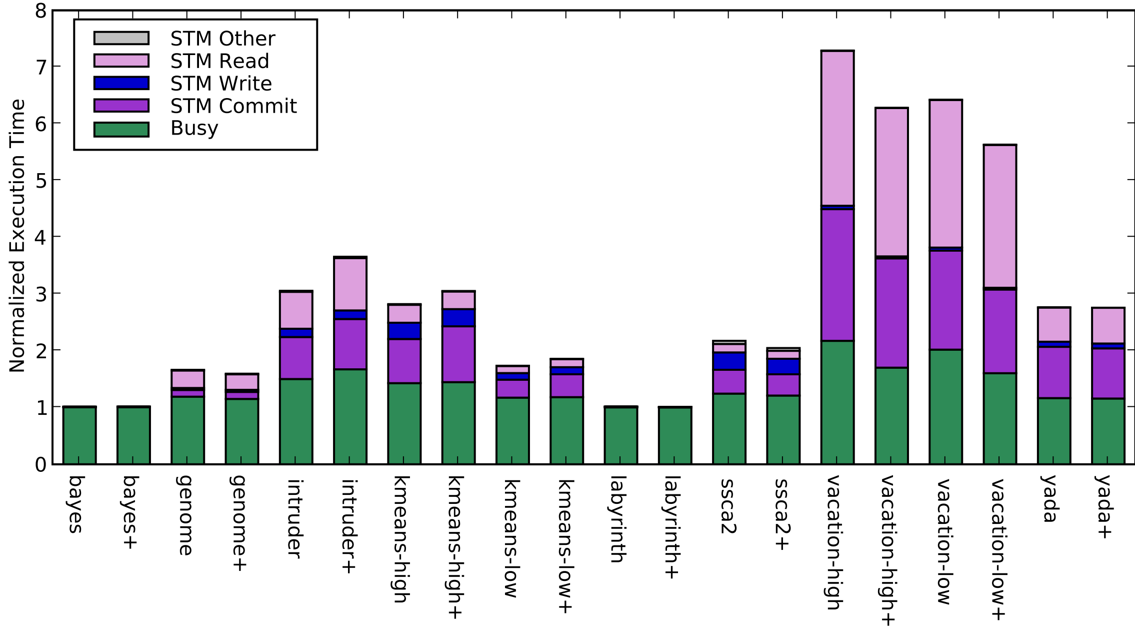


Figure 5.1: The lazy STM execution time breakdown for a single processor run. Execution time is normalized to that of the sequential code without transaction markers or read/write barriers.

Overall, the biggest bottleneck for STM is the maintenance and validation of the read set that occurs on each read barrier. Even after optimizations, the overhead is significant for transactions that read non-trivial amounts of data (e.g., `vacation`). In the lazy STM, the read barriers also have a secondary source of overhead. Since main memory is lazily updated by transactions, read barriers must first search the data versioning log for the most recent value. Fortunately, the software Bloom filter used by the lazy STM eliminates most unnecessary searches.

The other significant bottleneck occurs during the commit barrier. The lazy STM needs to perform three traversals of the write set: first to acquire locks, then to write the data versioning log to memory, and finally to release the locks. A pass must also be made over the read set to revalidate it. In comparison, the eager STM only needs to make one pass over the write set (to release its locks), but it still must perform a pass over the read set. Lock handling is expensive as it involves atomic instructions and causes additional cache misses and coherence traffic.

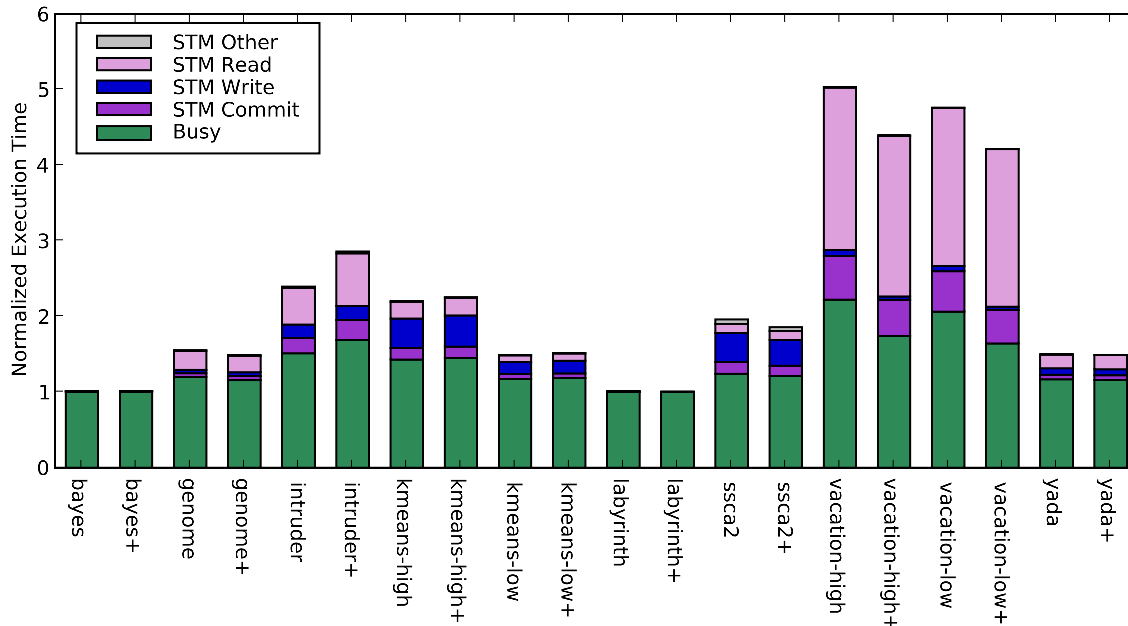


Figure 5.2: The eager STM execution time breakdown for a single processor run. Execution time is normalized to that of the sequential code without transaction markers or read/write barriers.

A third important factor (not shown in Figures 5.1 and 5.2) is that an STM transaction may not detect a conflict until it validates its read set during transaction commit. This is because a read by an ongoing transaction is not visible to any other transaction committing a new value for the same address. As a result, these invisible reads increase the amount of work wasted on aborted transactions.

Manual or compiler-based optimizations can be used to reduce STM overhead [1, 44]. For example, redundant barriers can be removed via common subexpression elimination or unnecessary barriers can be eliminated by identifying immutable or thread-local variables. Nevertheless, even optimized STM code must execute a barrier for each unique shared variable and a barrier to commit the transaction. A special case occurs for read-only transactions, in which read barriers do not need to be executed [32]. Unfortunately, read-only transactions are not the common case in most applications.

In HTMs, hardware support eliminates most overhead for transactional bookkeeping. No additional instructions are needed to maintain the read set or write set. Loads check the write set automatically for the latest values. Read set validation occurs transparently and continuously as coherence requests are processed. Thus, the number of cycles wasted on aborted transaction execution is minimized. Finally, only a single pass is required to apply the data versioning log to main memory. On the other hand, HTMs may experience performance challenges on transactions whose read set and write set overflow the available cache space or associativity. There are several proposed techniques that virtualize HTM systems using structures in virtual memory [27, 29, 71].

5.2 Strong Isolation

The differences between HTM and STM extend beyond performance. An important property for TM systems is whether they provide *strong isolation* for transactions. For a transaction to be strongly isolated, it must be isolated from *non-transactional* memory accesses in addition to transactional ones. With this property, a TM system provides a consistent ordering among transactional and non-transactional memory accesses, allowing programmers to predict how the transactional and non-transactional code blocks in their program will interact.

High-performance STM systems typically forsake strong isolation because it requires instrumentation of all memory accesses and thus adds to the already high STM overhead. Without strong isolation, STMs *may* fall prey to different dangers depending on the eager data versioning and conflict detection schemes implemented and on the interleaving of transactions and non-transactional memory accesses.

Figure 5.3 shows three scenarios that lead to unpredictable results with the lazy STM. The first scenario illustrates *non-repeatable reads*. Because Thread 2 does not use a write barrier, Thread 1's transaction cannot detect the read-write conflict on variable `x`. Thus, for the ordering shown, `t1` and `t2` will get different values, whereas the expectation is that they are equal. The second scenario is similar to the first, but shows how *lost updates* can occur. The two acceptable outcomes of the code result

Assume that initially x == y == 0

```
// Thread 1    // Thread 2
atomic {       ...
  t1=x;        ...
  ...          x=100;
  t2=x;        ...
}              ...
```

(a) Non-repeatable Reads

```
// Thread 1    // Thread 2
atomic {       ...
  t=x;         ...
  ...          x=100;
  x=t+1;       ...
}              ...
```

(b) Lost Updates

```
// Thread 1    // Thread 2
atomic {       ...
  x+=100;      ...
  y+=100;      ...
  ...          ...
}              t=x+y;
```

(c) Overlapped Writes

Figure 5.3: Isolation and ordering violation cases for the lazy STM system.

with `x` having the value 100 (Thread 1 before Thread 2) or the value 101 (Thread 1 after Thread 2). However, without strong isolation, Thread 1 will load the value 0 to `t`, not see the conflicting write from Thread 2, and then leave `x` with `t`'s incremented value of 1. The last example illustrates *overlapped writes* and has Thread 2 reading `x` and `y` without using read barriers while Thread 1 lazily commits its transactional stores. Thus, it is possible for Thread 2 to see the new value for `x` and the old value of `y` or vice versa.

The examples in Figure 5.3 can be dismissed as examples of data races that occur even if the transactions are replaced with locks. In [56], Larus and Rajwar present the simple race-free code example shown in Figure 5.4. In this program, Thread 1 uses a transaction to remove the head of a linked list and then uses the privatized value several times after the transaction. Meanwhile, Thread 2 uses a transaction to increment all the elements in the linked list. The two acceptable outcomes of this code are that either Thread 1 sees only the unincremented value when using `res.val` and Thread 2 operates on the shortened list (Thread 1 before Thread 2) or Thread 1 uses only the incremented value and Thread 2 operates on the entire list (Thread 1 after Thread 2). This simple program may run incorrectly on TM systems without strong isolation, but always works correctly if the transactions are replaced by locks.

The lack of strong isolation causes this code to behave unpredictably with all STM approaches [56]. As the two threads execute concurrently, `lhead` is included in the write set for Thread 1 and the read set for Thread 2. For TM systems with lazy data versioning, suppose that Thread 2 initiates its transaction commit, where it locks all the `.val` variables in the list, including `lhead.val`, and it verifies all variables

```
// Thread 1                    // Thread 2
ListNode res;                  atomic {
atomic {                           ListNode n = lhead;
    res = lhead;                   while (n != null) {
    if (lhead != null)                 n.val ++;
        lhead = lhead.next;            n = n.next;
}                                  }
use res.val multiple times;    }
```

Figure 5.4: The code for the privatization scenario that leads to unpredictable behavior on TM systems that do not provide strong isolation.

in its read set have not changed in memory, including `lhead`. While Thread 2 is copying its large write set, Thread 1 starts its commit stage. It locks its write set (`lhead`), validates its read set (`lhead` and `lhead.next`), and commits the new value of `lhead`. Subsequently, Thread 1 uses `res.val` outside of a transactional block as it believes that the element has been privatized by atomically extracting from the list. Depending on the progress rate of Thread 2 with copying its write set, Thread 1 may get to use the non-incremented value for `res.val` a few times before finally observing the incremented value committed by Thread 2.

The problem still occurs in TM systems with eager data versioning, but with a different interleaving of operations. Suppose that Thread 2 has already incremented `lhead.val` and is in the middle of incrementing all the elements of the list. At this point, Thread 2 has locks on all the `.val` fields it has incremented, including `lhead.val`. Meanwhile, Thread 1 starts its transaction, reads `lhead` and `lhead.next` and locks `lhead` before updating it. Thread 1 successfully validates its read set (`lhead` and `lhead.next`), commits its transaction, and begins using the incremented value of `res.val`. When Thread 2 wants to commit, it validates its read set and detects the update made to `lhead` made by Thread 1. Thread 2 then aborts its transaction and restores all the old unincremented `.val` fields of the list elements, including that of the original `lhead` that was removed by Thread 1 earlier. Consequently, Thread 1 now observes the unincremented value of when using `res.val`.

The privatization example in Figure 5.4 is not a unique case of race-free code that executes unpredictably without strong isolation. For example, the complementary

scenario, *publication*, where a thread makes a private object globally visible, can also lead to unpredictable results [86]. Another source of correctness issues is that many STM systems do not validate their read set until they reach the commit stage. Hence, it is possible to have a transaction that uses a pointer that has been modified in the meantime by another transaction. This series of actions can potentially result in an infinite loop or a memory exception [34].

Strong isolation can be implemented in an STM by using additional read and write barriers for non-transactional accesses memory accesses. The resulting performance degradation can be minimized by using compiler analysis to identify private data and data never accessed transactionally. In [86], Shpeisman et al. show their optimizations are able to reduce the overhead of strong isolation from 180% to 40% on a set of Java programs. Unfortunately, an overhead of 40% is still significant as it is in addition to the already high overhead experienced by STMs. Moreover, for workloads that use transactions more frequently or for unmanaged languages such as C++, the overhead will be higher. If strong isolation forces programmers to pick between performance and correctness, it has failed to deliver on the basic promise of TM: simple-to-write parallel code that performs well.

In contrast to STMs, HTM systems naturally implement strong isolation as all memory accesses, whether in transactions or not, are visible through the coherence protocol and can be properly handled. For the cases in Figure 5.3 (a) and (b), the write to `x` by Thread 2 on the HTMs will generate a coherence request for exclusive access. If Thread 1 has already read `x`, the coherence request will facilitate conflict detection and will cause Thread 1 to abort and re-execute its transaction. For the case in Figure 5.3.(c), once the transaction in Thread 1 is validated, it will generate NACKs to any incoming coherence request (shared or exclusive) for an address in its write set. Hence, Thread 2 will either read the new or old values for both `x` and `y`. The HTMs also correctly execute the privatization code in Figure 5.4 as Thread 1 cannot read partially committed state from Thread 2.

5.3 Signature-accelerated Transactional Memory

An ideal TM system would be one that combined the advantages of HTM (performance and semantics) with the advantages of STM (low cost and flexibility). *Signature-accelerated Transactional Memory (SigTM)*, is a hybrid TM system that uses small hardware signatures to reduce the runtime overhead of software transactions and transparently provide strong isolation and ordering guarantees. In more detail, SigTM features:

1. **High performance:** SigTM uses hardware signatures to conservatively represent the read set and the write set. Cache coherence messages are looked up in the signatures to provide conflict detection. Thus, much of the overhead is removed from the software barriers.

2. **Flexibility:** The only transactional policy that SigTM embeds in hardware is conflict detection. All other transactional functionality (such as data versioning and conflict management) is implemented in software and can easily be changed.

3. **Low cost:** SigTM requires no modifications to the hardware caches, which reduces hardware cost and simplifies support for features such as nesting and multithreaded cores.

4. **Strong isolation:** SigTM leverages the cache coherence protocol for conflict detection and thus can detect conflicts for both transactional and non-transactional memory accesses.

5.4 Hardware Signatures

In SigTM, each hardware thread has two signatures: one for transactional reads and one for the transactional writes. On read barriers, the read address is inserted into the read signature, and on write barriers, the written address is inserted into both the write signature and the software versioning log. Table 5.1 summarizes the instructions used by software to manage the signatures. Software can reset each signature, insert

Table 5.1: The user-level instructions for management of read and write signatures in SigTM.

Instruction	Description
`readSigReset` `writeSigReset`	Reset all bits in read set or write set signature
`readSigInsert` r1 `writeSigInsert` r1	Insert the address in register r1 in the read set or write set signature
`readSigMember` r1 r2 `writeSigMember` r1 r2	Set register r2 to 1 if the address in register r1 hits in the read set or write set signature
`readSigSave` r1 `writeSigSave` r1	Save a portion of the read set or write set signature into register r1
`readSigRestore` r1 `writeSigRestore` r1	Restore a portion of the read set or write set signature from register r1
`fetchExclusive` r1	Prefetch address in register r1 in exclusive state; if address in cache, upgrade to exclusive state if needed.

an address, check if an address hits in the signature, and save/restore its content. The last instruction in Table 5.1 allows software to prefetch or upgrade a cache line to one of the exclusive states of the coherence protocol (E for MESI). If the address is already in the M or E state, the instruction has no effect.

Each signature is Bloom filter [9] that uses a fixed-size register to record a set of addresses. To insert address A in a signature, hardware first truncates the cache line offset from the address. Next, it applies one or more hash functions on the remaining bits. Each function identifies one bit in the signature to be set to one. The insertion of an address into a signature is depicted in Figure 5.5. To check address A for a hit, the hardware truncates the address and applies the same set of hash functions. If all identified bits in the signature register are set, A is considered a signature hit. Figure 5.6 illustrates a series of basic operations on a Bloom filter.

During transaction execution in SigTM, cache coherence messages are looked up in the signatures to check for conflicts between transactions. A user-level configuration register per hardware thread is used to select if addresses from shared and/or exclusive requests should be looked up in the read set and/or write set signatures. If the enabled signature lookup returns a hit, the hardware signals a conflict and control is passed

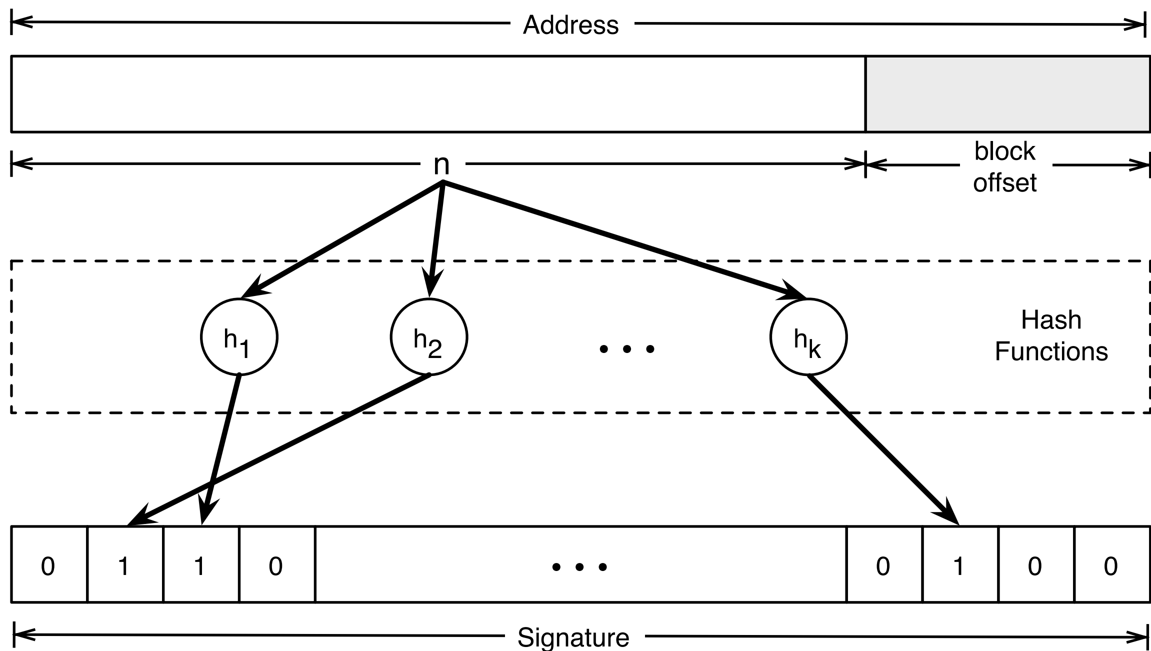


Figure 5.5: Inserting an address into a signature.

$$hash_1(x) = x \bmod 8$$
$$hash_2(x) = (x \oplus 2x) \bmod 8$$

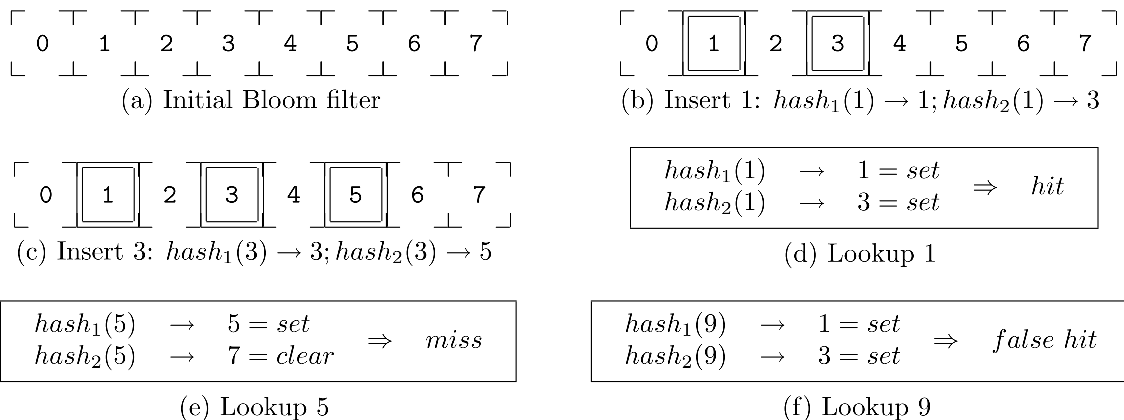


Figure 5.6: Operations on an 8-element Bloom filter with two hash functions. Dashed outlines indicate clear elements and solid outlines are set. Time progresses from (a) to (f). Note that aliasing of element 3 occurs in (c). In (e) a miss occurs because element 7 is not set. A false hit occurs in (f) because 9 was not inserted earlier.

to a preregistered user-level software handler. To avoid repeated invocations of the handler that prohibit forward progress, coherence lookups are temporarily disabled when the handler is invoked. The configuration register also indicates if coherence requests that hit in the write set signature should be acknowledged by the local cache or should receive a NACK reply.

Because of aliasing from its hash functions, Bloom filters can cause signature membership lookups to return false positives (but never false negatives). Note, however, that although false positives hurt performance, correctness is still assured because all true conflicts are detected by the signatures. The frequency of false positives can be reduced by increasing the signature length and adjusting the choice of hash functions. Furthermore, because the implementation of the Bloom filter is abstracted from the SigTM and application code, no recompilation of software is necessary when modifications to the hardware Bloom filters are made.

In summary, apart from hardware signatures and the NACK mechanism in the coherence protocol, SigTM requires no further hardware support. It is important to note that caches are not modified in any way.

5.5 Operation

To explain how SigTM operates, I will first describe a variant based on a lazy optimistic STM running on a system with broadcast coherence. Later in Section 5.8, I will explain how SigTM can be implemented using other STMs or systems. As one of many hybrid transactional memory designs, SigTM is unique because it presents a standalone and unified implementation. In other words, unlike other hybrid proposals [31, 55, 79], there is no backup STM, no switch between HTM and STM modes, and no fast versus slow code paths.

The SigTM hardware makes the global version clock and the software locks in the base lazy STM unnecessary. The software read set is also eliminated, but the data versioning is still done by software in order to buffer the write set until transaction commit. Contention management in SigTM is similar to that in the base STM, and it

retries aborted transactions after a randomized linear delay. Pseudocode summarizing the operations of software transactions in lazy SigTM is shown in Figure 5.7.

To start a transaction, `LazySigTMtxStart` (Figure 5.7, Line 1) requires two operations. First a hardware register checkpoint is made to allow a rollback if the transaction is aborted later. Second, lookups are enabled in the read signature for any exclusive cache coherence messages. If any of these lookups hit in the read signature, an exception is raised to indicate a conflict, and control is passed to the SigTM software. During transaction execution, cache coherence messages are not looked up in the write signature until transaction commit.

Accesses to shared variables in transactions are made using read and write barriers. `LazySigTMreadBarrier` (Line 5) first searches for the address is in the write set and uses the write signature as a quick test to avoid most unnecessary searches. If the address is not in the write set, the address is inserted in the read signature and then the value is read from memory. In `LazySigTMwriteBarrier` (Line 12) the address is inserted in the write signature and then the address and new value are added to the write set.

To commit a transaction, `LazySigTMtxCommit` (Line 16) has two phases: a validation phase followed by a writeback phase. In the validation phase, lookups of exclusive and shared coherence messages in the write signature are first enabled. A scan of the write set is then performed to call `fetchExclusive` on every written address. This validates the transaction by removing the corresponding cache line from other processors. Note that a `fetchExclusive` may replace the line brought by another `fetchExclusive` access without any correctness issue; correctness is achieved by removing the line from the caches of other processors and not by having a cached local copy. If any coherence messages hit in the write signature, a software handler is invoked that restarts the validation from scratch after a randomized backoff period. Coherence messages that hit in the read signature still cause the transaction to abort, however. After the `fetchExclusive` pass completes, the NACKing of any coherence requests that hit in the write signature is enabled. The validation phase then completes by clearing the read signature and disabling coherence lookups in the read signature.

```
 1: procedure LazySigTMtxStart
 2:     checkpoint()
 3:     enableReadSiglookup(EXCLUSIVE)
 4: end procedure

 5: procedure LazySigTMreadBarrier(addr)
 6:     if writeSigMember(addr) and writeSet.member(addr) then
 7:         return writeSet.lookup(addr)
 8:     end if
 9:     readSigInsert(addr)
10:     return Memory[addr]
11: end procedure

12: procedure LazySigTMwriteBarrier(addr, data)
13:     writeSigInsert(addr)
14:     writeSet.insert(addr, data)
15: end procedure

16: procedure LazySigTMtxCommit
17:     enableWriteSigLookup(EXCLUSIVE | SHARED)
18:     for all addr in writeSet do
19:         fetchExclusive(addr)
20:     end for
21:     enableWriteSigNack(EXCLUSIVE | SHARED)
22:     readSigReset()
23:     disableReadSiglookup()
24:     for all addr in writeSet do
25:         Memory[addr] ← writeSet.lookup(addr)
26:     end for
27:     writeSigReset()
28:     disableWriteSigNack()
29:     disableWriteSigLookup()
30: end procedure

31: procedure LazySigTMtxAbort
32:     readSigReset()
33:     disableReadSiglookup()
34:     writeSigReset()
35:     disableWriteSigLookup()
36:     doContentionManagement()
37:     restoreCheckpoint()
38: end procedure
```

Figure 5.7: Pseudocode for the basic functions in lazy SigTM.

Table 5.2: The minimum dynamic instruction counts for the lazy STM and lazy SigTM barriers. R and W represent the number of words in the transaction read set and write set, respectively.

	Read Barrier	Write Barrier	Commit Barrier
Lazy STM	41	40	$72 + 12R + 51W$
Lazy SigTM	8	27	$34 + 11W$

At this point, both the read set and write set are validated and the writeback phase can begin. A pass is made over the write set to update main memory with the contents of the data versioning buffer. After this completes, the transaction finally finishes committing by resetting the write signature and disabling any coherence lookups in and NACKs from the write signature.

5.6 Performance

SigTM executes the same number of barriers as STM; however, the overhead of SigTM's barriers is significantly lower as conflict detection is handled by hardware instead of software. Table 5.2 presents the minimum dynamic instruction counts for STM and SigTM barriers with lazy data versioning.

SigTM is able to significantly accelerate read barriers and commit barriers, the two biggest sources of overhead for STM. SigTM accelerates read barriers by replacing the software read set with a hardware read signature and by eliminating the need for time stamp and lock checks. The commit barrier overhead is reduced as the write set is traversed twice instead of three times and there is no need for read set validation in software. Moreover, instead of performing expensive atomic operations to acquire and release locks, SigTM uses the much faster `fetchExclusive` operation. Finally, SigTM does not provide a large performance improvement for the write barriers. It eliminates the need for time stamp and lock checks but still has to manage the software write set.

Another advantage of SigTM is that the coherence lookups in signatures during transaction execution allow a transaction to detect read conflicts when other transactions commit instead of when it commits itself. This continuous validation reduces the amount of doomed execution cycles because doomed transactions restart sooner.

SigTM's performance challenge comes from the admission of false positives when performing conflict detection. When these occur, transactions waste useful work by unnecessarily aborting. False positives can occur if a program's memory access patterns interact negatively with the hash functions used to implement the signatures. However, since the signatures' implementation is abstracted from software, the signatures' lengths can be increased and hash function adjusted in future generations to reduce the likelihood of false positives. An additional source of false conflicts occurs because SigTM leverages the cache coherence protocol for conflict detection and thus performs conflict detection at cache line granularity. HTMs also perform conflict detection at line granularity, but STMs typically employ word or object granularity.

5.7 Strong Isolation

To achieve strong isolation, a TM system must detect and handle non-transactional accesses to its read and write sets in addition to transactional accesses. By using its signatures, SigTM provides strong isolation and ordering guarantees for software transactions without additional instrumentation of code outside of transactions. Both non-transactional and transactional writes generate coherence messages that cause lookups in the signatures and signal conflicts if necessary.

Non-transactional writes to the read set are handled by looking up exclusive coherence requests in the read signature. This eliminates *non-repeatable reads* and *lost updates* (Figure 5.3, (a) and (b), respectively). In both cases, the non-transactional write performed by Thread 2 generates an exclusive coherence message that hits in the read signature of Thread 1. This causes Thread 1 to abort and retry its transaction, which leads to one of the expected final outcomes. This continuous read set validation also prevents SigTM from executing on inconsistent state that would lead

to infinite loops or memory faults. Finally, non-transactional reads to the read set do not need to be handled as they do not create conflicts.

SigTM's write signature takes care of non-transactional reads and non-transactional writes to the write set. The *overlapped writes* example in Figure 5.3 and the privatization example in Figure 5.4 result in unpredictable behavior because a non-transactional read is allowed to view only part of the transaction's modifications to memory. In SigTM, this problem is fixed by having the write signature NACK any coherence requests that hit in it during the *writeback* phase of transaction commit. Note that NACKing requests for an address is equivalent to holding a lock on it and can cause serialization and performance issues. SigTM uses NACKs only during write set copying, which is typically a relatively short event.

5.8 Alternative Implementations

This section discusses the implementation of SigTM using alternative STMs or multi-core systems.

5.8.1 Lazy vs. Eager Data Versioning

Thus far, I have discussed the design of SigTM based on a lazy STM, but SigTM can also be built on top of an STM with eager data versioning. In an STM with eager data versioning, writes within a transaction update memory directly and the original values are kept in an undo log in case the transaction needs to rollback. SigTM implements data versioning in software, so no hardware changes need to be made to implement an eager variant of SigTM.

In eager SigTM, the signatures are used slightly differently than in the lazy variant. Figure 5.8 presents pseudocode for all the basic functions in eager SigTM. Cache coherence messages need to be looked up in the write signature in addition to the read signature throughout the duration of the transaction. Any coherence requests that hit in the write signature are NACKed. Eager SigTM's read barrier is the same as in lazy SigTM, but in the write barrier, eager SigTM must execute `fetchExclusive`

Table 5.3: The minimum dynamic instruction counts for the eager STM and eager SigTM barriers. R and W represent the number of words in the transaction read set and write set, respectively.

	Read Barrier	Write Barrier	Commit Barrier
Eager STM	32	57	$51 + 15R + 6W$
Eager SigTM	4	32	15

after inserting the address in the write signature and before updating the undo log and memory. Any `fetchExclusive` instructions that time out because of NACKs cause the transaction to abort (pessimistic conflict detection). During transaction abort, the NACKs from the write signature guarantee atomic and isolated restoration of original values from the undo log. Finally, transaction commit is very simple in eager SigTM and just requires resetting the read signature and write signature. Table 5.3 presents the minimum dynamic overhead for STM and SigTM barriers with eager data versioning.

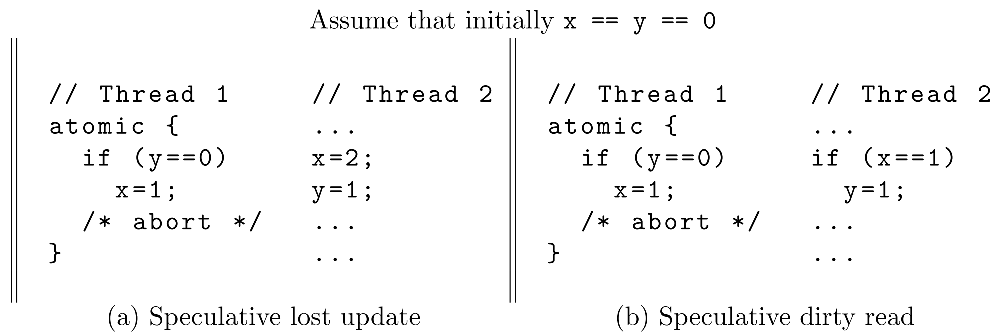
Figure 5.9 illustrates two cases that can lead to unpredictable behavior on eager STMs without strong isolation: *speculative lost update* and *speculative dirty read* [86]. In the *speculative lost update* example, the expected value for `x` is either 1 or 2, but the shown interleaving results in an incorrect value of 0. Similarly, the *speculative dirty read* results in `x` with the value 0 instead of 1. Both of these errors occur because the memory accesses to `x` by Thread 2 break the isolation and atomicity of the eager STM's transaction abort. In SigTM, however, the NACKs from the write signature guarantee that rolling back a transaction appears isolated and atomic. This also guarantees predictable behavior for the privatization example in Figure 5.4 as non-transactional accesses that interfere with the undo log are NACKed by the write signature.

5.8.2 Line vs. Object Granularity Conflict Detection

Thus far, SigTM has been described with cache line granularity conflict detection, which works well with languages like C. For object-oriented languages, however, a

```
 1: procedure EagerSigTMtxStart
 2:     checkpoint()
 3:     enableReadSiglookup(EXCLUSIVE)
 4:     enableWriteSiglookup(EXCLUSIVE | SHARED)
 5:     enableWriteSigNack(EXCLUSIVE | SHARED)
 6: end procedure

 7: procedure EagerSigTMreadBarrier(addr)
 8:     readSigInsert(addr)
 9:     return Memory[addr]
10: end procedure

11: procedure EagerSigTMwriteBarrier(addr, data)
12:     writeSigInsert(addr)
13:     fetchExclusive(addr)
14:     writeSet.insert(addr, Memory[addr])
15:     Memory[addr] ← data
16: end procedure

17: procedure EagerSigTMtxCommit
18:     readSigReset()
19:     disableReadSiglookup()
20:     writeSigReset()
21:     disableWriteSigLookup()
22: end procedure

23: procedure EagerSigTMtxAbort
24:     readSigReset()
25:     disableReadSiglookup()
26:     for all addr in writeSet do
27:         Memory[addr] ← writeSet.lookup(addr)
28:     end for
29:     writeSigReset()
30:     disableWriteSigNack()
31:     disableWriteSigLookup()
32:     doContentionManagement()
33:     restoreCheckpoint()
34: end procedure
```

Figure 5.8: Pseudocode for the basic functions in eager SigTM.



(a) Speculative lost update (b) Speculative dirty read

Figure 5.9: Isolation and ordering violation cases for the eager STM system.

better match would be object granularity conflict detection since object-oriented programmers reason about their program behavior in terms of objects. To accomplish object granularity conflict detection, only two simple software changes need to be made in SigTM. First, the object header of objects should be inserted into signatures instead of the addresses of all the object fields. Second, the software data versioning log must be adjusted to support object granularity.

In addition to semantic advantages, conflict detection at object granularity offers performance benefits when compared to line granularity conflict detection. With the former, read barriers and write barriers become less frequent as they only need to be executed once per object instead of once per each field in that object. This leads to smaller SigTM software overhead. Moreover, since fewer addresses are inserted in the signatures, signature lookups that return false positives are less probable. They can still occur because of aliasing from the signature hash functions or because of multiple object headers occupying the same cache line, but their frequency is nevertheless reduced.

Finally, the SigTM hardware can also support a mixed environment that uses both line and object granularity conflict detection. Line granularity can be advantageous for large objects like multi-dimensional arrays since only a small subset of the array elements may be accessed at a time in each transaction. Thus, for these kinds of objects, line granularity can help reduce the number of true conflicts. For other data types, object granularity is advantageous for the reasons previously discussed.

5.8.3 Broadcast Coherence vs. Directories

Up till now, the description of SigTM has assumed a broadcast protocol that makes cache coherence messages visible to all processors in the system. The snooping coherence used in current multi-core systems meets this assumption. Broadcast coherence allows the hardware signatures to view all coherence requests—even for cache lines that have been accessed by a transaction but subsequently replaced in the local cache. Larger-scale multi-core systems may use directories to filter coherence traffic. To implement SigTM correctly in such systems, one can use the LogTM sticky directory states [65]. In LogTM, cache replacement does not immediately modify the directory state. Hence, the directory will continue to forward coherence requests to a processor even after it evicts a line from its caches.

5.9 System Issues

To be a complete approach for implementing transactional memory, SigTM needs to handle several system issues. In this section, I will explain how SigTM handles transaction nesting, hardware multithreading, thread suspension and migration, virtual memory paging, and inter-process isolation.

5.9.1 Nesting

The existing hardware in SigTM can support an unbounded number of transaction nesting levels. Unlike HTMs and other hybrid designs [7, 31, 41, 55, 65, 79, 88], the hardware cost for SigTM does not increase with the number of nesting levels supported. Moreover, since data versioning is implemented in software, SigTM simply has to separate the management of the software write sets of the parent and child transactions and then merge them appropriately. Support for transaction nesting is important as transactions in applications are likely to call library code that may also be parallelized with transactions.

SigTM provides conflict detection for nested transactions [61] by saving and restoring signatures at transaction boundaries. To begin a nested transaction, SigTM must

first save the contents of the read signature and write signature. The signatures are not reset, and the child transaction continues inserting addresses in the signatures so that the signatures contain addresses from both the parent and child transactions. If the signature detects a conflict, all the saved parent signatures must be checked in order to determine how many nesting levels to roll back. Proceeding from the youngest to the oldest parent nesting level, the corresponding signatures are examined and the first one that does not result in a hit indicates the point for rolling back. When a child transaction commits, it needs to properly merge with the parent transaction. For closed nesting, the saved parent signatures are discarded and execution proceeds with the current contents of the signatures. On the other hand, if an open-nested transaction commits, the parent's signatures are restored to indicate that the child transaction committed independently.

5.9.2 Hardware Multithreading

For processor cores that support multiple hardware threads, SigTM requires a separate set of signatures and configuration registers per thread. Since the signatures are maintained outside the cache, it is relatively straightforward to add this extra hardware. However, since one hardware thread may fetch a line that is later accessed by another hardware thread, signature lookups on misses and coherence upgrades are no longer sufficient. Stores to lines in modified or exclusive states must also be looked up in the signatures of other threads. Moreover, if any thread currently has its write signature in NACK mode, load hits to lines in modified or exclusive states must also be looked up in the write set signatures of other threads.

5.9.3 Thread Suspension and Migration

Because of interrupts or thread scheduling by the operating system, SigTM must be able to suspend execution of a transaction. Handling interrupts in a transaction can be a relatively expensive operation, but an efficient way to accomplish this task is the approach used by the XTM system [29]. If the interrupt is not critical, this approach first waits a short period of time to see if a transaction finishes executing. Since many

transactions are relatively short, one is likely to complete in this short wait period. However, if the interrupt is critical, the system checks to see if there are any relatively young transactions executing. If so, that transaction is aborted as not much work will be lost by rolling it back. By handling the interrupt in one of these two manners, no transaction state needs to be saved or restored, which leads to lower overhead.

If neither short nor young transactions are currently executing, SigTM must suspend an arbitrary transaction. To achieve this, nothing special needs to be done for the data versioning log since it is implemented in software. However the contents of the thread's read signature and write signature must be saved by the operating system. To provide conflict detection for all suspended transaction, the operating system can use one additional pair of signatures. The combined signatures (bitwise OR) of all the suspended read signatures and write signatures are stored in the respective additional signature. When the additional signature detects a conflict, software processing is required to determine which suspended thread to abort. Finally, to resume a transaction, the operating system simply restores that transaction's signatures and recalculates the contents of the additional signatures for the transactions that are still suspended. This technique of transaction suspension and resumption can also be used to achieve the migration of a transaction between two cores.

5.9.4 Paging

Swapping virtual memory pages to and from disk poses a challenge for SigTM because physical addresses are recorded in the signatures. To properly provide conflict detection for transactions in the presence of page remapping, two things must be done. First, the operating system must keep track of the threads that access each page. Then, when paging occurs, all the physical addresses from the new mapping must be inserted into the signatures. This approach will ensure correctness as no true conflicts are missed, but it has the disadvantage of an increased probability of false conflicts. To reduce the probability of false conflicts, a second set of signatures can be maintained using virtual addresses. When updating the physical signatures after

a page swap, the virtual signatures can be used as filters that indicate which physical addresses should be inserted from the new mapping.

5.9.5 Inter-process Isolation

Because SigTM uses physical addresses to populate its signatures and to perform conflict detection, it is possible for memory references from different processes to interfere with each other. For example, a malicious process running on a machine could purposely saturate its signatures. This would cause its signatures to generate conflicts or NACKs for all remote memory references and while it would not affect correctness, it would prevent other processes from making forward progress.

There are several ways in which SigTM can prevent this type of denial-of-service attack. One solution is to tag all coherence messages with an address space identifier. SigTM would then only generate conflicts or NACKs if the remote coherence request hits in the signature *and* the address space identifier matches. The second requirement prevents false conflicts between processes. Another solution is to rely on the operating system. Because the operating system is continuously monitoring the progress of all threads, it can detect when a thread is repeatedly aborted or NACKed. If this behavior occurs, the operating system can adjust its thread scheduling policies so that two conflicting processes do run at the same time.

5.10 Related Work

In the past few years, there has been significant research activity on transactional memory, covering topics such as implementation techniques, programming constructs, runtime systems, and contention management. In this section, I will discuss some alternative hybrid transactional memory proposals as well as related work on signature based HTMs and strong isolation. For a thorough coverage of all aspects of transactional memory research, I refer the reader to Larus and Rajwar [56].

5.10.1 Hybrid Transactional Memory

The first hybrid TM systems were proposed to address the virtualization challenges of HTM [31,55]. These systems combine an HTM with an STM implementation, switching from the former to the latter on context switches or when hardware resources become exhausted. These early hybrid TMs introduce modifications to caches (for the HTM) and require two versions of the code for every transaction. In contrast, SigTM does not require changes to the caches, and it only requires one version of the code.

Subsequent hybrid transactional memory schemes introduced hardware changes to address performance bottlenecks of software transactions [79, 88]. Of these two approaches, SigTM is closest in design to the HASTM system [79]. In HASTM, software-controlled *mark bits* are added to each cache line and are used to create filters that eliminate redundant read or write barriers and read set validations during transaction commits. However, because cache lines that are part of the read or write set can be replaced, HASTM cannot rely exclusively on mark bits for conflict detection and must be able to fall back to the slower STM bookkeeping.

In contrast to HASTM, SigTM is a standalone system without a backup mode as its signatures can encode read and write set membership even if their sizes exceed the local cache capacity. However, because aliasing may occur in its signatures, SigTM cannot use its signatures to eliminate redundant read or write barriers or unnecessary read set validations during commit. SigTM must also do extra work to correctly maintain its signatures for events such as thread suspension and virtual memory paging. On the other hand, HASTM can discard the contents of its filters on such events and still correctly provide hardware acceleration of software transactions. Unfortunately, introducing support for hardware multithreading and nested transactions in HASTM can be expensive as it requires multiple sets of mark bits per cache line.

The other hybrid TM that adds hardware to accelerate software transactions is RTM [88]. RTM accomplishes this by modifying the local caches to support its *alert on update (AOU)* mechanism and by adding 5 states to the MESI coherence protocol to facilitate its *programmable data isolation (PDI)* feature. In contrast, SigTM does

not require cache modifications and only needs a NACK feature to be added to the coherence protocol.

Recent hybrid transactional memory proposals have been able to provide strong isolation for software transactions without the addition of extra barriers in non-transactional code regions. The first of these was SigTM, which accomplishes strong isolation through the use of signatures and the cache coherence protocol. The hybrid TM system by Vallejo et al. [94] is also able to solve the privatization problem illustrated in Figure 5.4. It achieves this by combining an HTM with an STM that provides pessimistic concurrency control (PCC) [89]. Unlike other hybrid designs that combine HTM with STM, this implementation only requires one code path. However, the implementation only works with eager data versioning and as it does not provide strong isolation for all scenarios with non-transactional memory references (e.g., publication).

A very recent hybrid TM system that provides strong isolation is the User Fault-On (UFO) system [7]. The UFO system associates two bits with each cache line throughout the memory hierarchy to enable read and write protection modes. The bits indicate whether the line was read or written in a transaction and are used to generate a memory fault if an invalid non-transactional access is made to a marked line. A disadvantage of the UFO system is that its technique for providing strong isolation has high overhead when transaction nesting occurs as all the pages in the read and write sets must be visited to copy the added protection bits. Moreover, because the UFO system combines HTM with STM, it requires two versions of the application code.

5.10.2 Signature-based HTMs

SigTM was inspired by the Bulk HTM, which was the first TM system that used signatures for conflict detection [23]. As an HTM, the Bulk design requires additional hardware to implement lazy data versioning in caches and thus must deal with cache capacity limitations. In contrast, SigTM implements data versioning in software and

requires no hardware support beyond the read set and write set signatures. LogTM-SE is similar to Bulk but implements eager data versioning [98]. It requires additional hardware to implement the undo log, including an array of recently logged cache lines.

5.10.3 Strong Isolation

Several researchers have observed that strong isolation is necessary for predictable behavior in TM systems [13,56,86]. In particular, Shpeisman et al. [86] have categorized the problematic cases for both eager and lazy TM systems that do not provide strong isolation. They also presented a compiler methodology, including optimizations, that use additional barriers in non-transactional code in order to provide strong isolation guarantees. In [13], Blundell et al. claim that there are cases in which strong isolation leads to unexpected results. However, they are simply observing that transactions can replace lock-based synchronization in some cases (atomicity) but not in others (coordination).

Chapter 6

Evaluating Signature-accelerated Transactional Memory

In designing SigTM, my goal was to create a hybrid TM system that combined the advantages of HTMs and STMs. In the previous chapter, I qualitatively described how SigTM is a design that has high-performance, flexibility, cost-effectiveness, and strong isolation. Of these four characteristics, two can be verified quantitatively: high-performance and cost-effectiveness. In this chapter, I use STAMP to quantitatively analyze these two aspects of the SigTM design.

6.1 Methodology

To evaluate SigTM, I used the same simulator-based approach described in Section 4.1. Table 6.1 summarizes all the parameters of the simulated system. As before, the processor model assumes an IPC of 1 for all instructions that do not access memory, but all memory hierarchy timings are modeled.

The only hardware difference from before is the addition of signature registers to each core. The signature hash functions are described in Table 6.1 and each can set/lookup any of the bits in the signature register (i.e., "true Bloom signatures" as named by [82]). Detailed analyses of hash functions suitable for hardware signatures

Table 6.1: Configuration for the simulated multi-core system with SigTM support.

Feature	Description
Processors	1 to 32 x86 cores, in-order, single-issue
L1 Cache	64KB, private, 4-way associativity, 32B line, 1-cycle access Provides TM bookkeeping for HTM systems
Network	32B bus, split transactions, pipelined, MESI
L2 cache	8MB, shared, 32-way associativity, 32B line, 12-cycle access
Memory	100-cycle off-chip access
Signatures	32 to 2048 bits per signature register
SigTM Hash Functions	(1) unpermuted cache line address (2) cache line address permuted as in [23] (details below) (3) address from (2) shifted right by 10 bits (4) permutation of cache line address (details below)

Bit permutations used in hash functions (bit indices, LSB is 0):
(2): [0-6, 9, 11, 17, 7-8, 10, 12, 13, 15-16, 18-20, 14]
(4): [25-24, 22-21, 18-17, 13-11, 1-0, 20, 10 16, 19, 2, 6, 4, 23, 7-8, 5, 26, 9, 15-14, 3]

can be found in [22, 23, 74, 81], and the selection of the hash functions used in SigTM was based on these previous works.

For the experiments in this chapter, four TM system designs were implemented on top of the simulator:

- **Lazy STM:** An x86 port of the TL2 software TM system [32] that is described in Section 2.5.1. It performs lazy versioning using a software write buffer. To provide conflict detection, it uses locks for data in the write set during commit. Conflicts for data in the read set are detected by checking version numbers periodically, and after a transaction aborts three times, the lazy STM uses a randomized linear backoff mechanism. Finally, it detects conflicts at word-granularity and provides weak isolation of transactions.

- **Eager STM:** An eager version of TL2 as described in Section 2.5.2. Similar to the McRT STM [80], it uses an undo log and holds locks on data in the

write set throughout the transaction to provide versioning. Conflict detection is similar to the lazy STM system, and its conflict management, conflict detection granularity, and weak isolation policies are the same.

- **Lazy SigTM:** A hybrid version of the lazy STM that follows the SigTM system [17] as described in Section 5.5. It uses hardware signatures to track the read and write set and to implement fast conflict detection. Data versioning is still in software, and the lazy STM's contention management policy (randomized linear backoff) is used. Finally, conflict detection is performed at line granularity and strong isolation of transactions is provided.

- **Eager SigTM:** The eager equivalent of the lazy SigTM as described in Section 5.8.1. Conflict detection utilizes the hardware signatures but the software for data versioning uses an undo log. Eager SigTM uses the same contention management policy as lazy SigTM (and the lazy and eager STMs) and provides line granularity conflict detection and strong isolation of transactions.

To evaluate the two SigTM designs, I used the STAMP benchmark suite to analyze SigTM's performance and to quantify SigTM's hardware cost. Since I used a simulator for the analysis, only twenty STAMP input data sets and configurations (the variants without the '++' suffix) were used.

6.2 Performance Analysis

The goal of SigTM was to accelerate software transactions by adding a small amount of simple hardware to reduce the software overhead described in 5.1. To evaluate how well SigTM achieves this goal, I compared the performance of four TM systems. The performance of lazy SigTM was compared to that of lazy STM, and a similar comparison was made between eager SigTM and eager STM. No direct comparisons between lazy and eager TM systems were made because the purpose of the experiments to see how effective SigTM accelerates software transactions and not to determine the best overall design for a TM system. Figures 6.1 and 6.2 present the speedups of the two

lazy TM systems and two eager TM systems, respectively. The number of cores is scaled from 1 to 32, and higher speedups are better. For these experiments, SigTM uses 2 Kbits per read and per write signature.

To provide further insight into performance issues, Figures 6.3 and 6.4 show the execution time breakdown for the runs with 16 processors on the lazy TM systems and eager TM systems, respectively. Execution time is broken into "busy" (useful instructions and cache misses), "conflict" (time spent on aborted transactions), and "imbalance" (workload imbalances). To show the overhead of software transactions, I separate time spent on read and write barriers and commit from busy time. Miscellaneous barriers, like those starting a software transaction, are accounted for in the "other" segment. Lower execution times are better.

Figures 6.1 and 6.2 show that in general, SigTM scales similar to STM but that SigTM's relative performance is often much better. On average, at 32 cores, lazy SigTM is 2.05× faster than lazy STM and eager SigTM is 1.45× faster than eager STM. Applications that match this average behavior are `genome`, `kmeans-low`, and `vacation`. The explanation for eager SigTM's smaller performance improvement is relatively straightforward. In comparison to the lazy STM, the eager STM's read, write, and commit barriers have much smaller overhead (compare Table 5.2 and Table 5.3). Thus, when accelerating lazy STM, SigTM has more opportunities for improving performance than when accelerating eager STM. For both lazy and eager SigTM, however, Figures 6.3 and 6.4 show that SigTM is effective in reducing the overhead of read, write, and commit barriers.

The one application where STM outperforms SigTM is `bayes`. In this application, several threads concurrently modify a global array that has multiple elements per cache line. As explained in Section 5.6, SigTM cannot perform conflict detection at finer granularity than cache lines. In comparison, the STMs detect conflicts at word-granularity. Thus, in `bayes` the modifications of the global array cause transactions in SigTM to abort more than in STM, and Figures 6.3 and 6.4 confirm that SigTM loses more performance to violations than STM. This false-sharing behavior also appears in `vacation`. However, for this application, SigTM's acceleration of the large number of read barriers outweighs the performance loss from coarser-grain conflict detection.

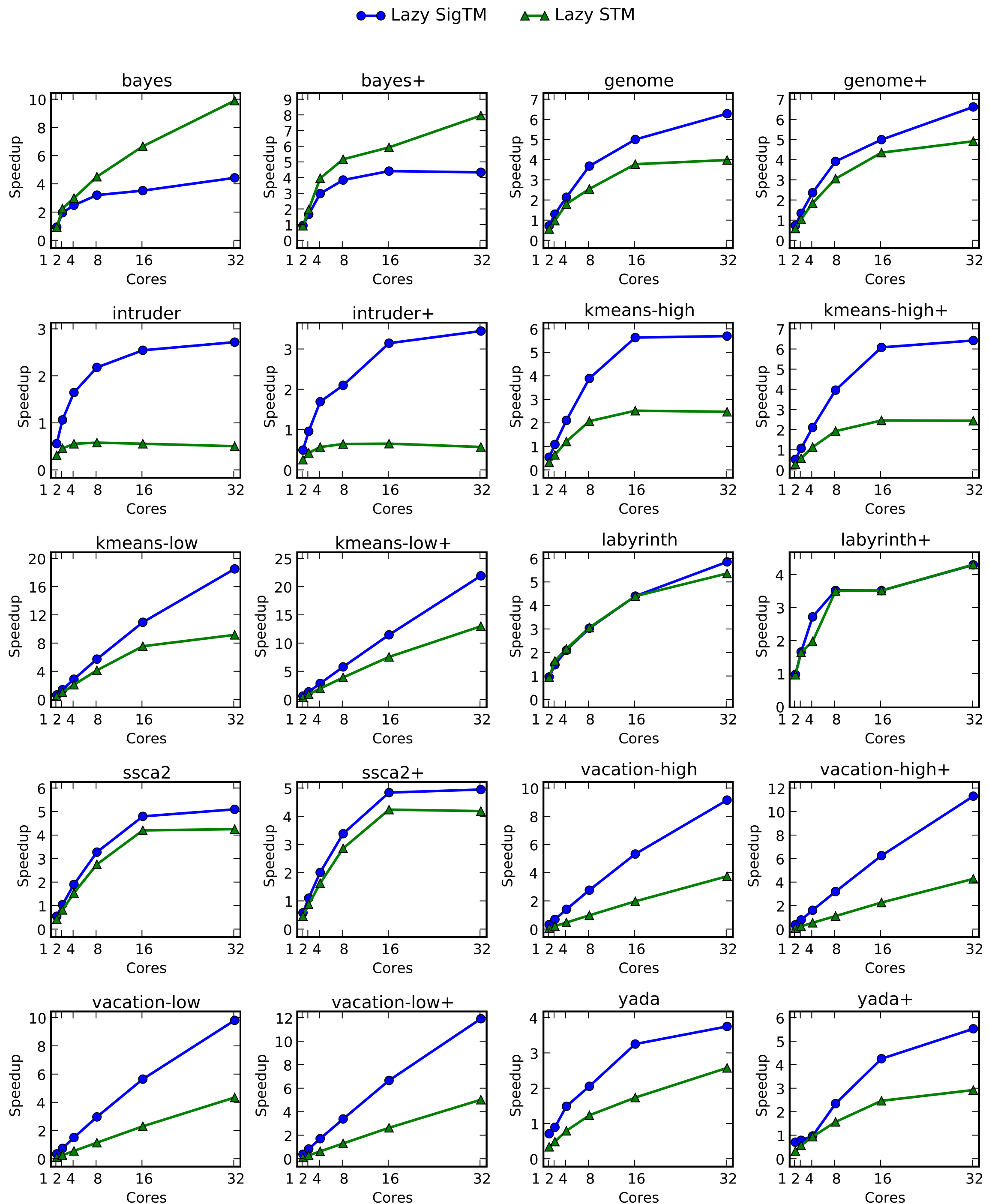


Figure 6.1: STAMP speedups over sequential code on lazy SigTM and lazy STM. Application variants that use the larger data set are indicated by a '+' appended to the application name. The suffixes *-low* and *-high* indicate variants with low and with high contention, respectively.

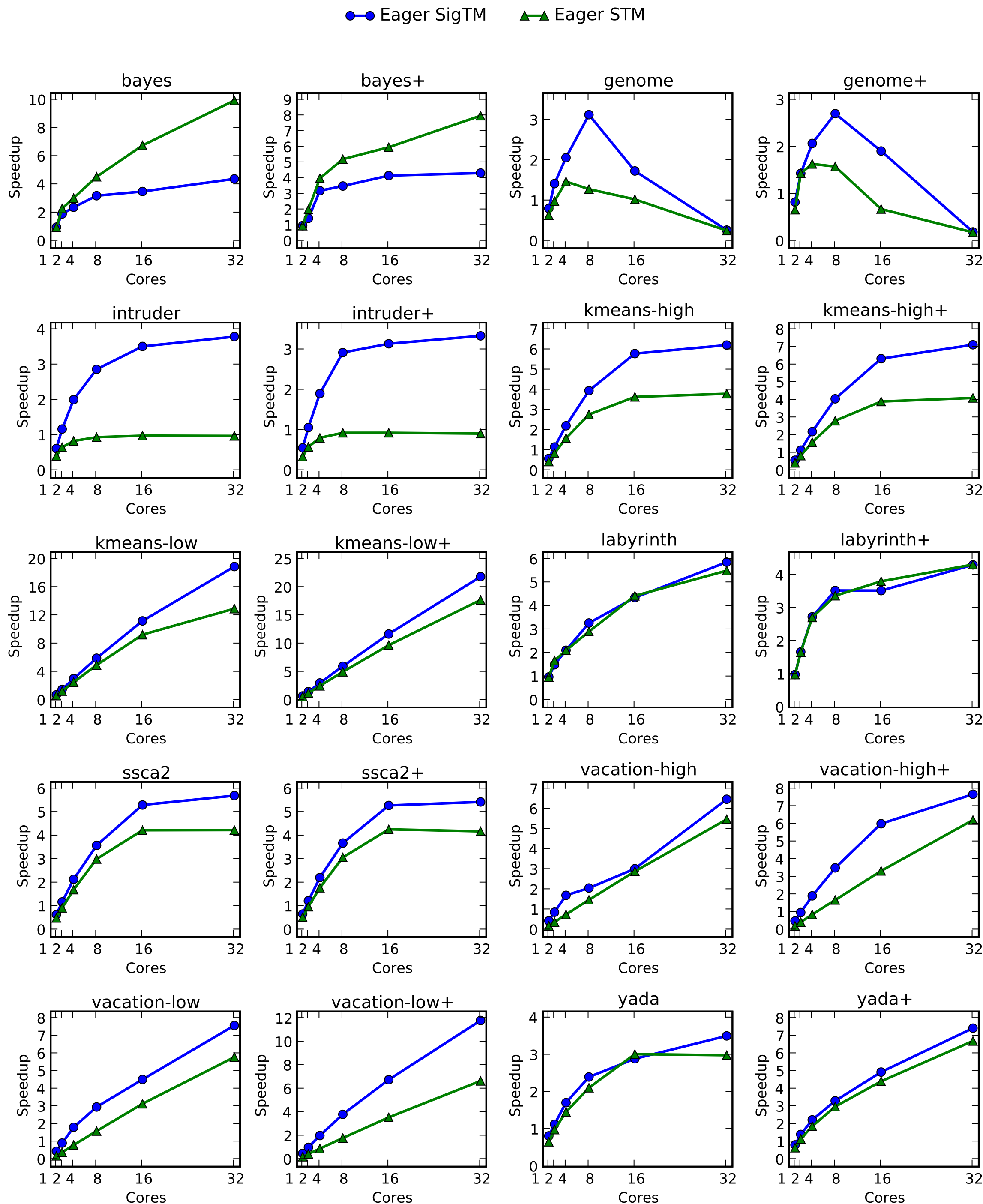


Figure 6.2: STAMP speedups over sequential code on eager SigTM and eager STM. Application variants that use the larger data set are indicated by a '+' appended to the application name. The suffixes *-low* and *-high* indicate variants with low and with high contention, respectively.

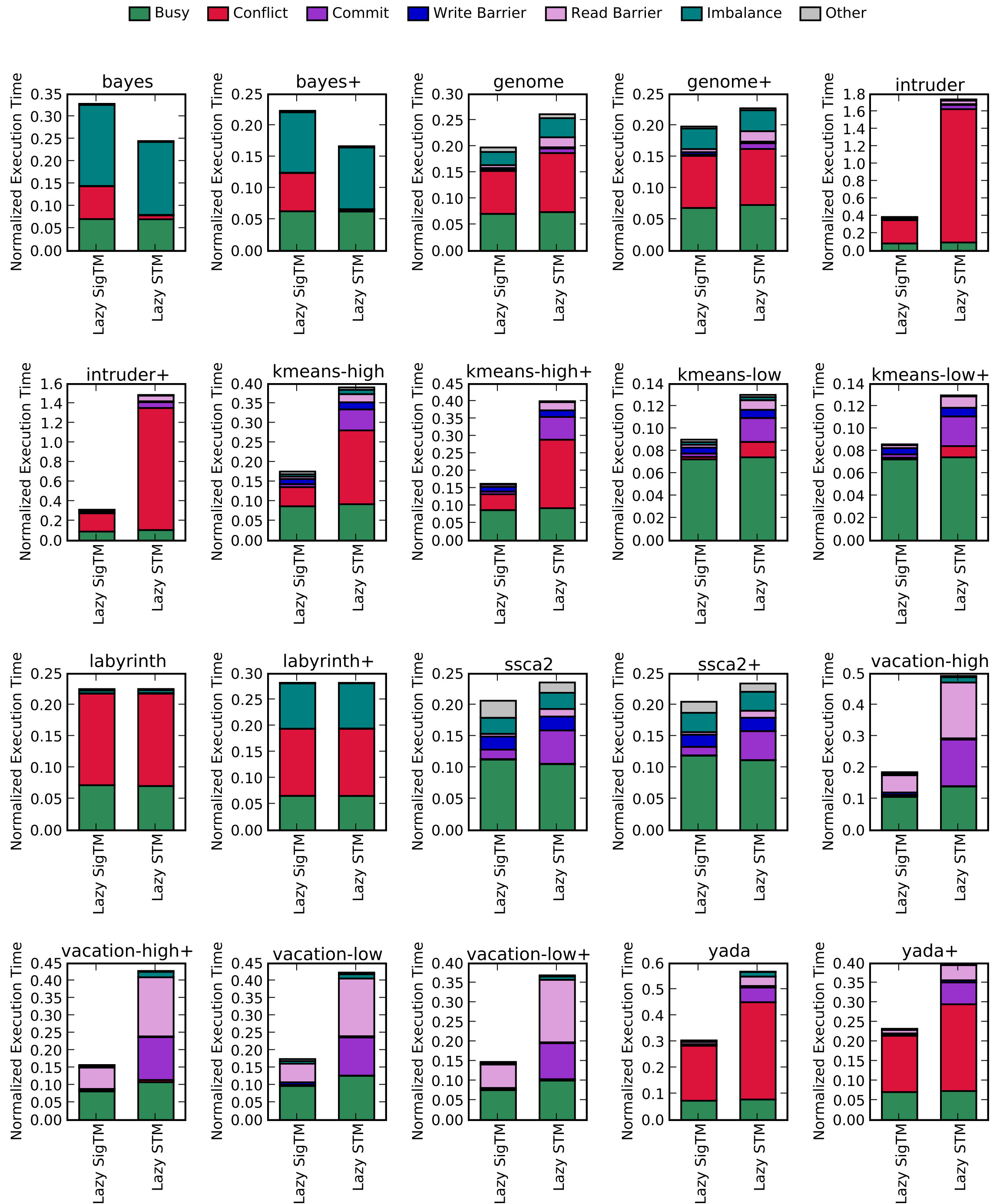


Figure 6.3: The execution time breakdown for 16-processor runs on lazy SigTM and lazy STM. Execution time is normalized to that of the sequential code without transaction markers or read/write barriers.

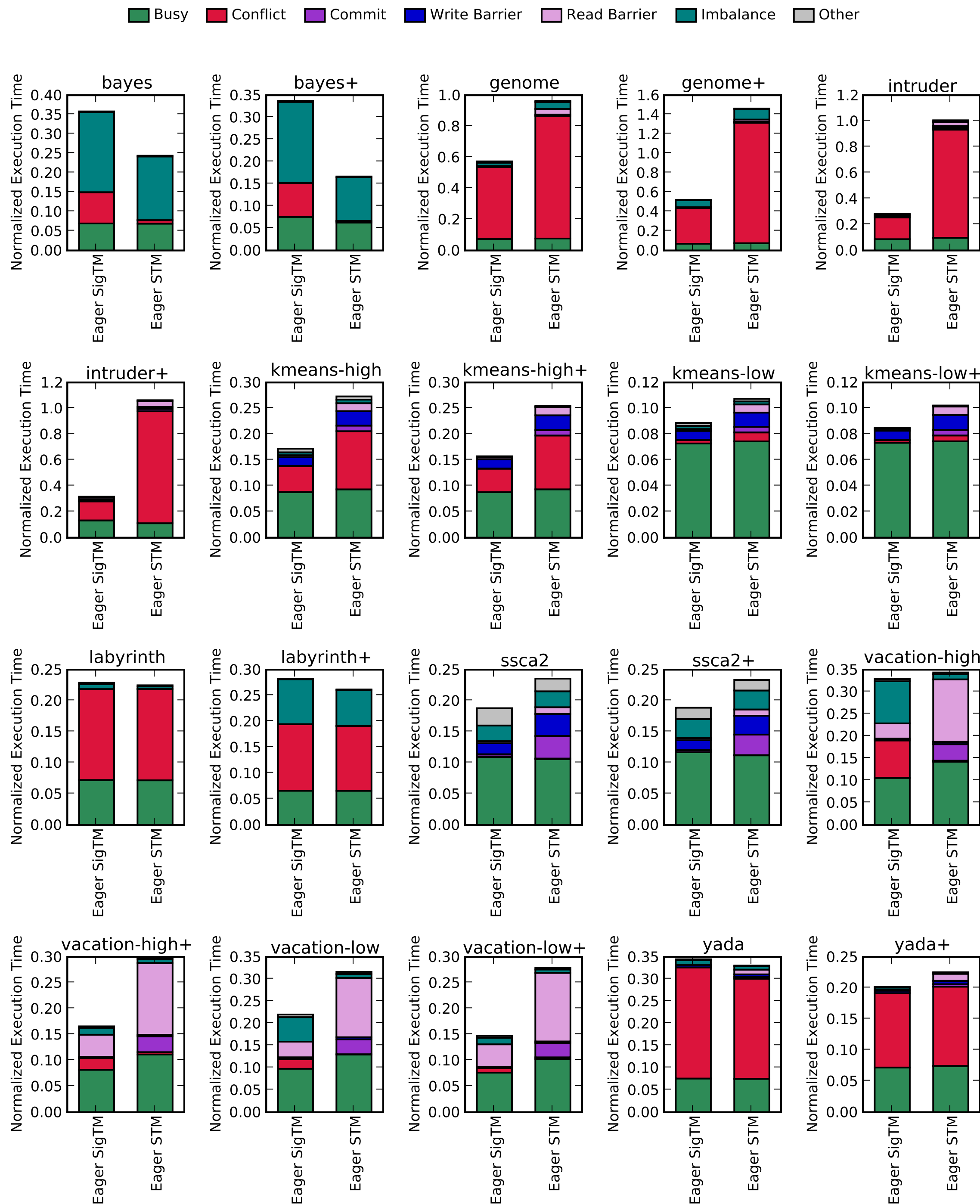


Figure 6.4: The execution time breakdown for 16-processor runs on eager SigTM and eager STM. Execution time is normalized to that of the sequential code without transaction markers or read/write barriers.

Of the STAMP applications, the two that exhibit the largest performance difference between SigTM and STM are `intruder` and `kmeans-high`. SigTM is able to achieve a performance improvement of almost 6× for `intruder` and almost 4× for `kmeans-high`. Among the STAMP applications, these two have relatively large numbers of read and write barriers with respect to the lengths of their transactions. Consequently, when SigTM reduces the overhead of read and write barriers, the lengths of transactions in these two application becomes much shorter. This reduces the amount of useful work that can be potentially lost to violations and also reduces the time duration where a transaction is vulnerable to a violation. Thus, SigTM is able to perform much better than STM on these two applications, and Figures 6.3 and 6.4 confirm that SigTM spends much less time in violation.

There are also two STAMP applications that do not have a large performance difference between SigTM and STM. In `labyrinth`, transactions are very long, which means that the overhead of read, write, and commit barriers is easily amortized. `ssca2` also does not have much overhead from software transactions as it has very few read and write barriers. Since the difference between SigTM and STM is in the overhead of these barriers, applications where these are not the main bottleneck will have very similar performance between SigTM and STM.

Finally, some of the STAMP benchmarks demonstrate performance differences between lazy and eager variants of SigTM. The most notable of these is `genome`. As with the eager STM and HTM, the pessimistic conflict detection scheme used by eager SigTM is disadvantageous for this workload because it does not guarantee forward progress. Thus, with more than 8 cores, eager SigTM no longer scales because the higher rates of conflicts cause livelock. In contrast, lazy SigTM continues to scale as optimistic conflict detection guarantees forward progress.

The other applications where eager SigTM performs differently than its lazy variant are: `kmeans`, `vacation`, and `yada`. In these three benchmarks, the performance difference between eager SigTM and eager STM is much smaller than that between lazy SigTM and lazy STM. Because eager STM's barrier overhead is smaller than that of lazy STM, the SigTM hardware provides less acceleration in the eager variant than in the lazy one. This effect is not seen in all the STAMP applications, as some

other factors (e.g., high contention, livelock, etc.) may be the dominating factor in overall performance.

6.3 Hardware Cost Analysis

The SigTM results in Section 6.2 assume long read set and write set signatures (2 Kbits per signature) that eliminate virtually all false conflicts due to aliasing from the signature hash functions. Figures 6.5, 6.6, 6.7, and 6.8 present the normalized performance of SigTM with 16 processors as I vary the read and write signature lengths from 2 Kbits down to 32 bits. Higher performance is better, and an operating point with both high performance and a short signature length is desired. The same set of hash functions is used in all experiments (see Table 6.1).

6.3.1 Read Signature Cost Analysis

The graphs in Figures 6.5 and 6.6 show that SigTM's performance is highly sensitive to the read signature length. Between the lazy and eager variant of SigTM, the performance with varying read signature lengths is similar because the read signature is used the same way in both variants; coherence requests are looked up in the read signatures throughout the entire duration of a transaction in both systems. Varying the read signature length, however, does show that the resulting behavior of the STAMP applications divides them into two groups: a group that is relatively sensitive to read signature length and a second group that is relatively insensitive.

In the first group are `bayes`, `genome`, `intruder`, `vacation`, and `yada`. The performances of these applications all drop significantly with read signature lengths of less than 2 Kbits and at 32 bits, all perform at less than 20% of that with 2 Kbits. `yada` has the largest performance hit among the four applications, and at 1024 bits, it drops to 40% of its performance. All of these applications have moderate to long transactions and medium to large numbers of read barriers, which result in a high sensitivity to read signature length. First, long transactions make aborts very costly in terms of performance. Since reducing the signature length increases false positives

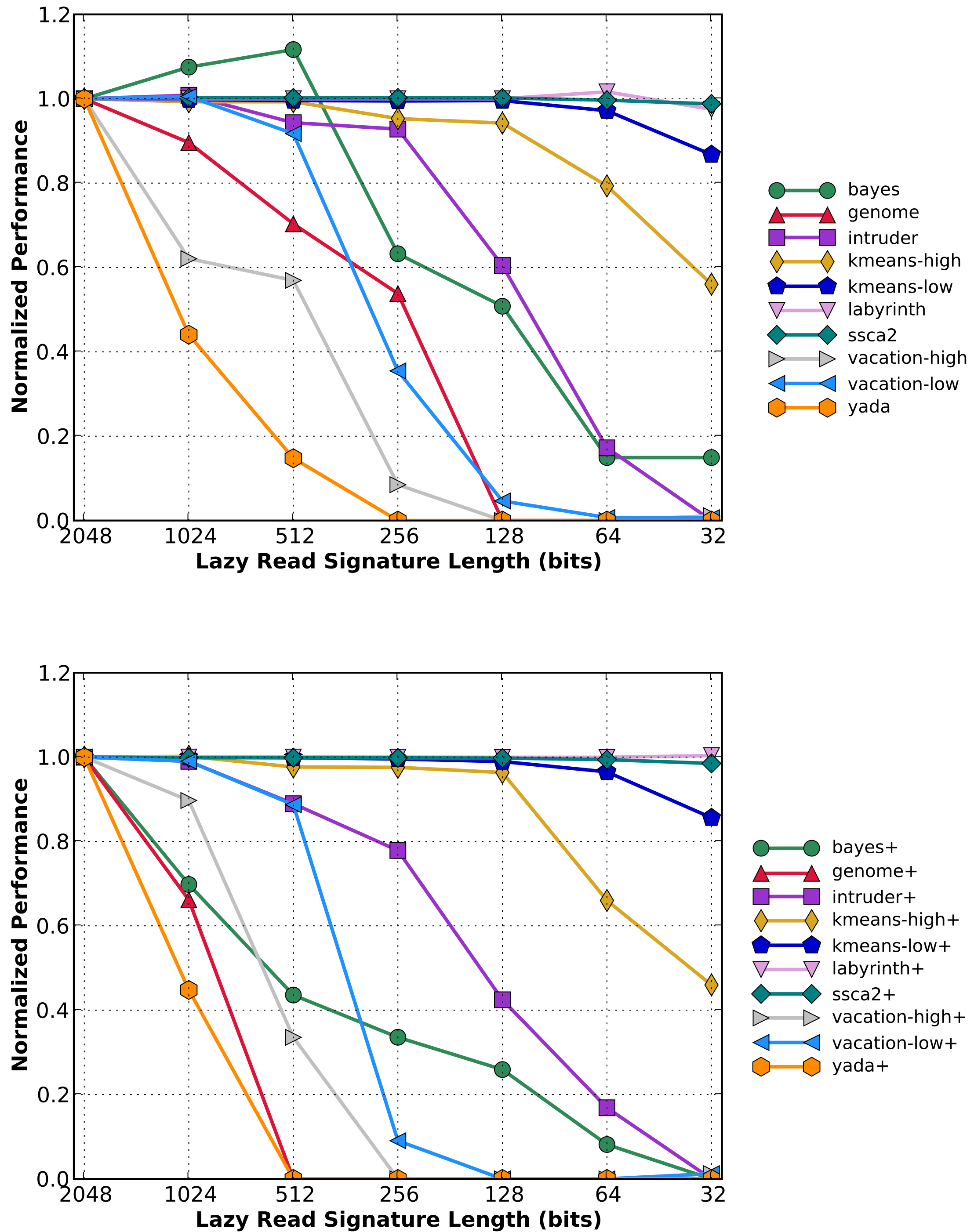


Figure 6.5: The effect of read signature length on lazy SigTM performance (16 processor runs). Note that bit count, and therefore accuracy, *decreases* from left to right.

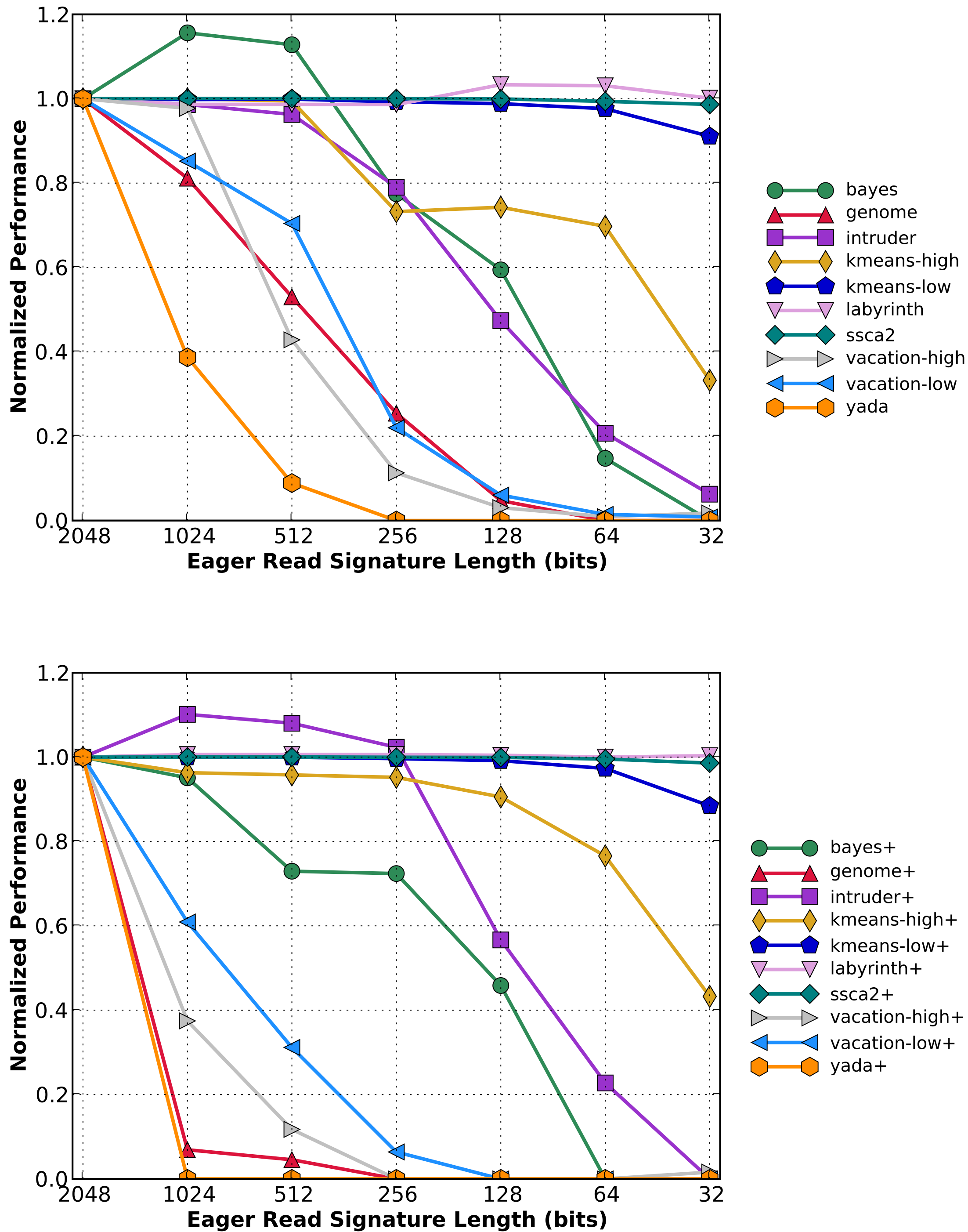


Figure 6.6: The effect of read signature length on eager SigTM performance (16 processor runs). Note that bit count, and therefore accuracy, *decreases* from left to right.

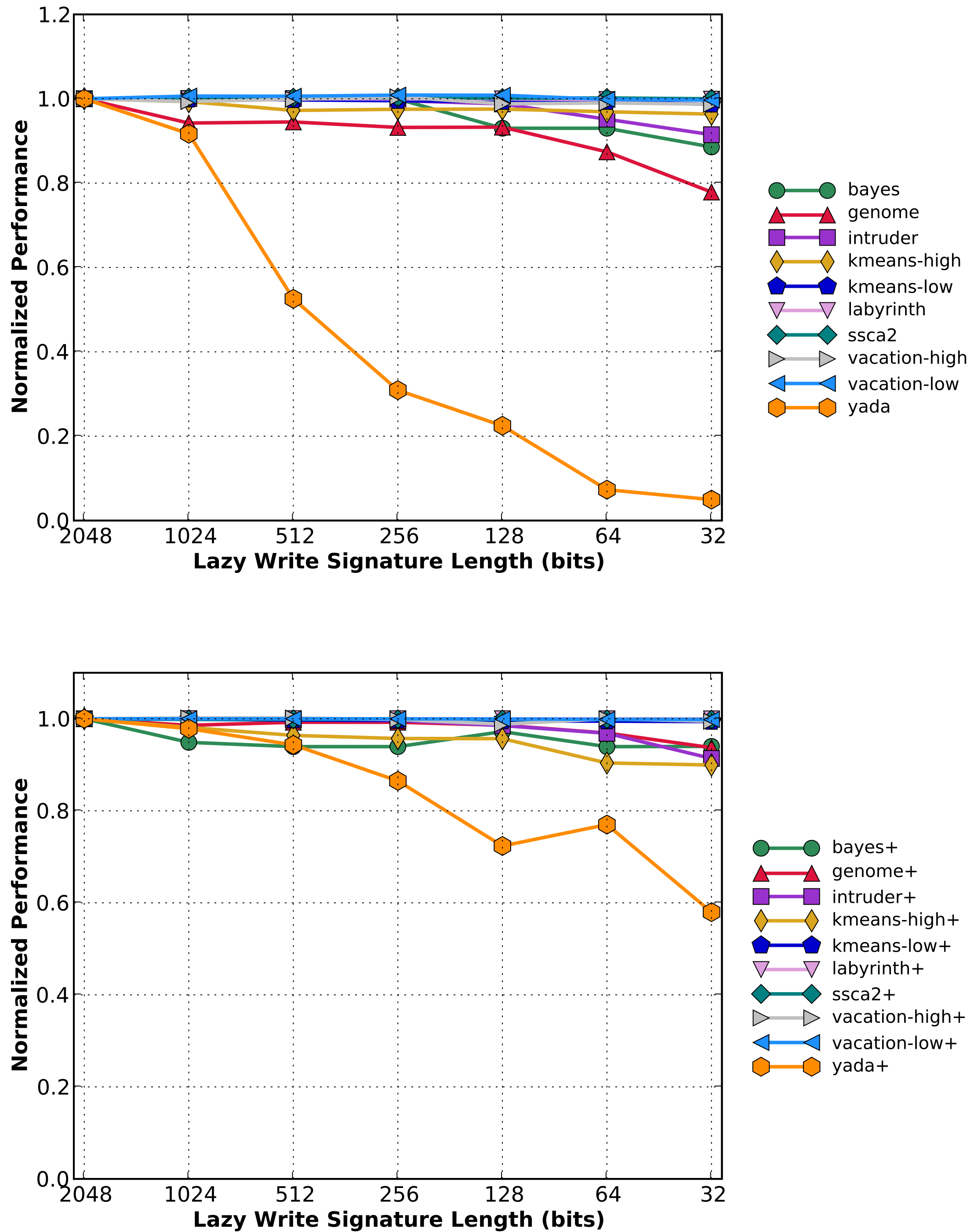


Figure 6.7: The effect of write signature length on lazy SigTM performance (16 processor runs). Note that bit count, and therefore accuracy, *decreases* from left to right.

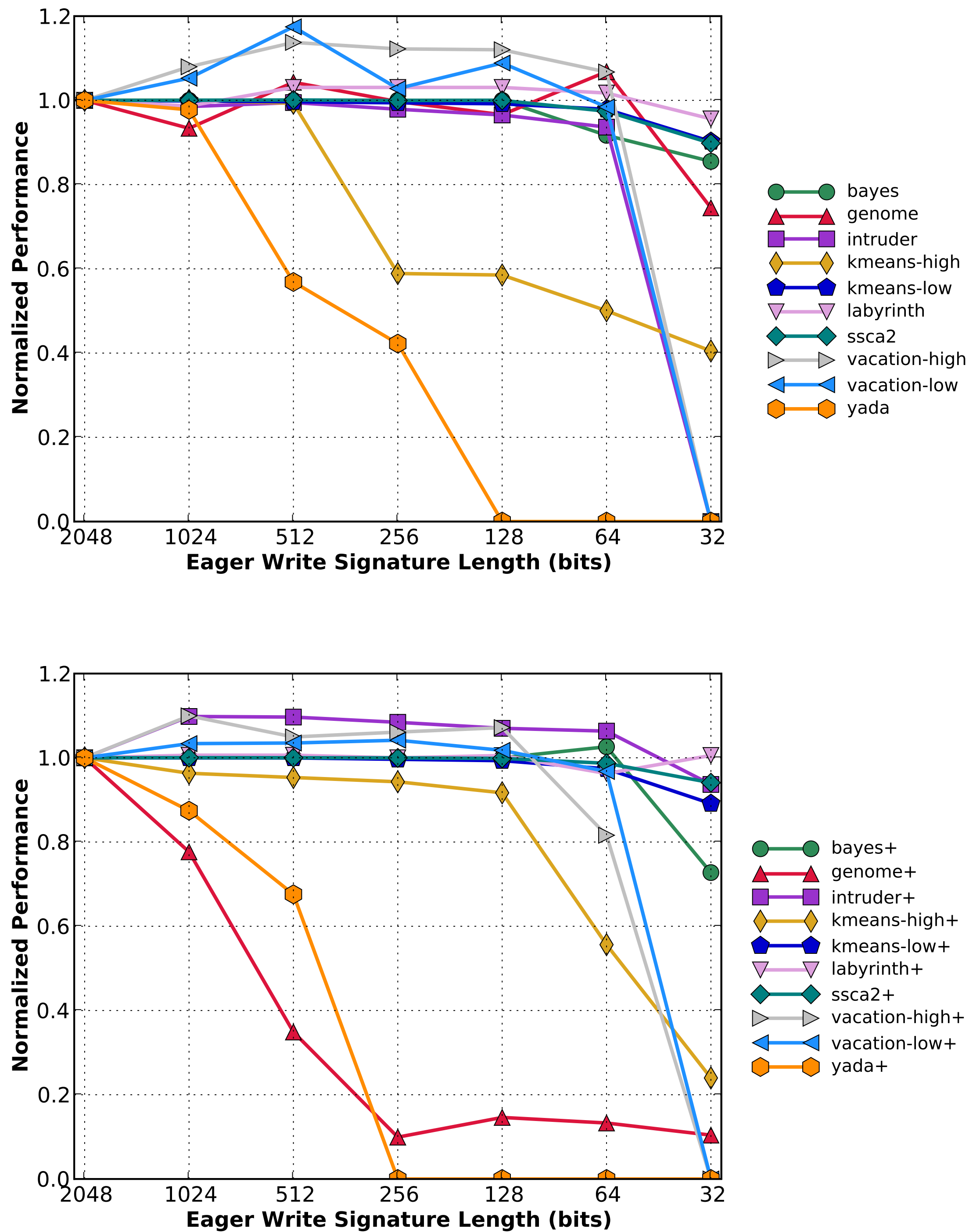


Figure 6.8: The effect of write signature length on eager SigTM performance (16 processor runs). Note that bit count, and therefore accuracy, *decreases* from left to right.

and hence the total number of aborts, these five applications quickly succumb to performance degradation from shorter signatures. Second, frequent read barriers make it more likely for false conflicts to occur as they cause more bits in the signature to be set. This is especially true for `vacation` and `yada` as their large numbers of read barriers per transaction (from 256 to 608), mean that a 1024 bit Bloom filter can have up to a 70% chance of false positives [82].

It is important to note that depending on the hash functions used and the memory access patterns exhibited, large numbers of read barriers do not necessarily lead to more false conflicts. This behavior is seen in `labyrinth`, as each read barrier accesses a unique cache line (see Section 3.4.5). Since one of the signature hash functions is the unpermuted cache line address, false conflicts will only start occurring in `labyrinth` for read signature lengths close to to the numbers of read barriers per transaction in `labyrinth` (35 to 46 read barriers). Thus, this benchmark is able to perform well even with short signatures of 32 bits.

Finally, there is a second group of benchmarks whose performance is relatively insensitive to read signature length: `kmeans`, `labyrinth`, and `ssca2`. These applications have relatively few read barriers per transaction, which makes them less prone to false conflicts caused by reduced signature lengths. Moreover, `kmeans` and `ssca2` also have relatively short transaction lengths, which means that transaction aborts are relatively less expensive. The exception to this is `labyrinth`, which has very long transactions, but still performs well with short signatures (for the reasons discussed earlier). Because of these factors, at 512 bits, these applications still perform within 95% of their original performance with 2 Kbits, and all but `kmeans-high` are within 80% at 32 bits.

6.3.2 Write Signature Cost Analysis

Figure 6.7 shows that, with the exception of `yada`, none of the STAMP applications exhibit particular sensitivity to the write signature length in lazy SigTM. With 128 bit signatures, all workloads but `yada` still perform within 90% of their original performance with 2 Kbit signatures on lazy SigTM. Of the STAMP applications, `yada` has

the highest number of write barriers per transaction (108 write barriers) by an order of magnitude. Thus, with any read signature length less than 1 Kbits, the large number of write barriers in `yada` make it succumb to false conflicts.

Performance on lazy SigTM is relatively insensitive to write signature length because of two reasons. First, these applications have low numbers of write barriers per transaction; thus, their write sets can be accurately represented even with short signatures. Second, because of lazy versioning, write signatures detect conflicts only during the commit stage to guarantee write isolation as the write set is copied to memory. Since the commit stage is short, the write signature accuracy is less critical than that of the read signature, which is used for coherence lookups throughout the entire duration of the transaction.

On the other hand, Figure 6.8 indicates that the performance of SigTM with eager versioning is more sensitive to the write signature length. This is because eager SigTM enables coherence lookups in the write signature throughout the transaction execution and not only during transaction commit like lazy SigTM. Consequently most of the applications on eager SigTM need a longer write signature length of 512 bits in order to maintain performance within 95% of that with 2 Kbit write signatures. Two of the applications, `genome` and `yada` actually require an even longer write signature length of 1024 bits to maintain 75% performance. `genome` is very prone to livelock (see Section 4.3.2), and the additional false conflicts caused by write signature lengths shorter than 2 Kbits exacerbate this. Finally, in the case of `yada`, the application's larger number of write barriers make it more susceptible to false conflicts from shorter write signatures.

6.3.3 Cost Sensitivity to Number of Cores

It is important to determine if the number of cores affects the signature cost analysis presented earlier. To test this, I selected three of the STAMP applications that were representative of different sensitivities to signature length: `labyrinth` (low sensitivity), `genome` (medium sensitivity), and `yada` (high sensitivity). By varying the

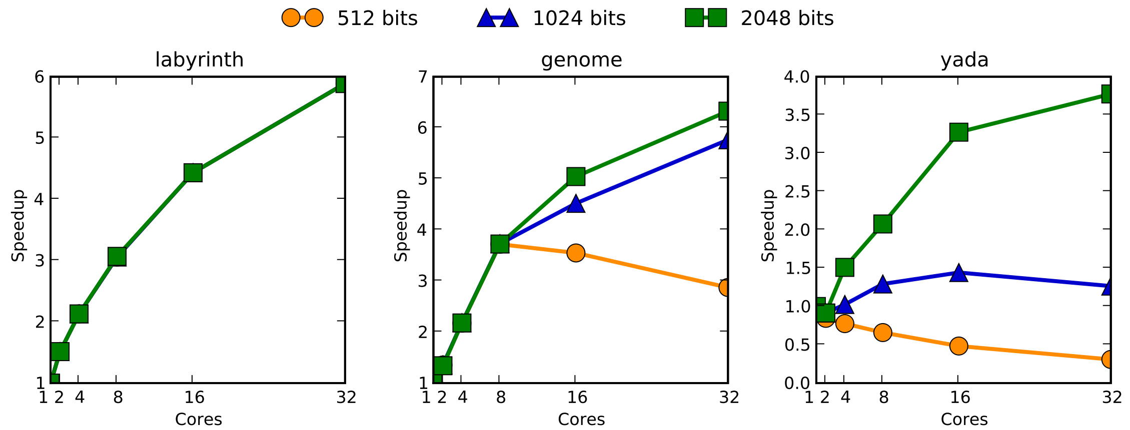


Figure 6.9: The effect of read signature length on lazy SigTM scalability.

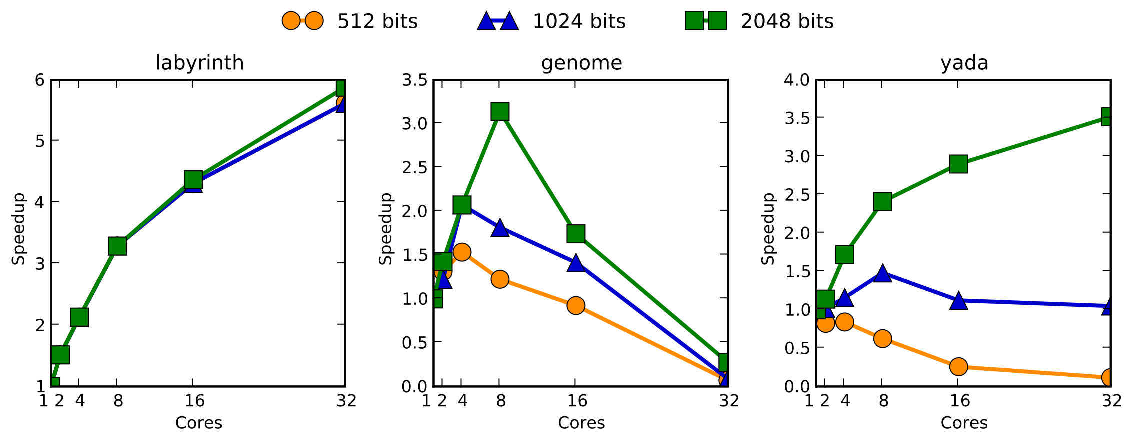


Figure 6.10: The effect of read signature length on eager SigTM scalability.

number of cores from 1 to 32, the speedup relative to sequential code without software transaction overhead was measured for signature sizes of 512, 1024, and 2048 bits. Figures 6.9 and 6.10 show how the read signature length affects scalability on lazy and eager SigTM, respectively. The corresponding data for write signatures are shown in Figures 6.11 and 6.12.

As expected, the higher the sensitivity to signature length, the more scalability suffers as read signature length is reduced. For applications with high sensitivity (`vacation` and `yada`), any read signatures shorter than 2 Kbits fail to scale to 32 cores. In fact, reducing the signature length by a factor of two results in a performance

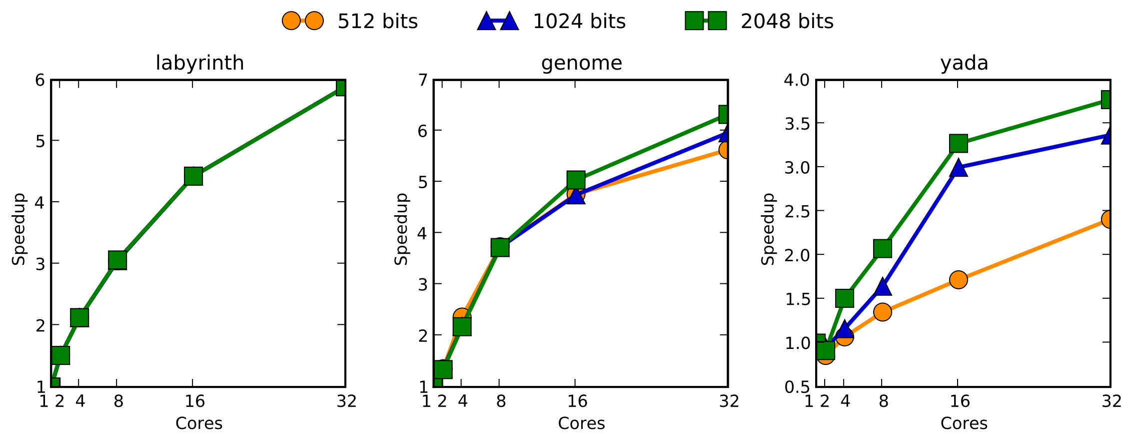


Figure 6.11: The effect of write signature length on lazy SigTM scalability.

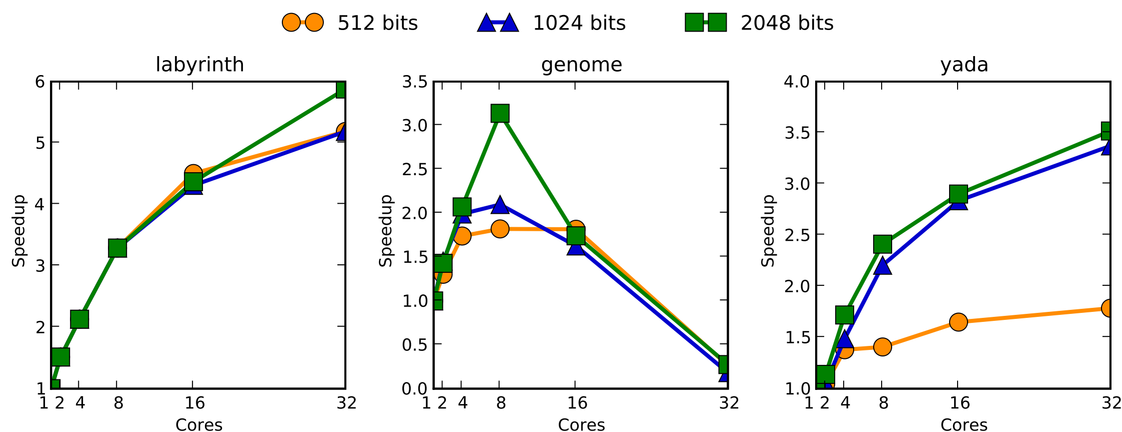


Figure 6.12: The effect of write signature length on eager SigTM scalability.

degradation of 3× at 32 cores. The medium sensitivity category consists of `bayes`, `genome`, and `intruder`, and these applications still scale with 1 Kbit read signatures (though they fail to scale with 512 bits). Note that as discussed before, `genome` on eager SigTM suffers from livelock, which explains why none of the three curves for `genome` scale in Figure 6.10. Finally, the low sensitivity group (`kmeans`, `labyrinth`, and `ssca2`) still scale well when the read signature length is reduced.

The general results for the varying write signature lengths are the same as for the varying read signature lengths with the exception of `yada`. Figures 6.11 and 6.12 show that between 1 Kbit and 2 Kbit write signatures, `yada` has a performance difference

of less than 20%, whereas for the read signatures there was a difference of 3×.. Note that even with 512 bit write signatures, `yada` is able to scale to 32 cores (though its absolute performance is about 2× worse than that with 1 Kbit or 2 Kbit write signatures). Overall, across the STAMP applications there is greater variance among read set sizes than among write set sizes, which leads read signature lengths to have a greater impact on scalability than write signature lengths.

6.3.4 Cost Analysis Summary

Given the applications and system sizes I studied, my recommendation is 1 Kbit read signatures and 128 bit write signatures for lazy SigTM and 1 Kbit read signatures and 512 bit write signatures for eager SigTM. Hence, the per hardware thread cost of SigTM is 1152 bits and 1526 bits of storage for lazy and data versioning, respectively, plus the logic for the hash functions.

Compared to previous work, my results differ slightly. Using a set of Java workloads, the Bulk HTM recommended read and write signatures of 2 Kbits each [23]. Based on the STAMP workloads, SigTM's recommended read and write signature lengths are shorter than those for Bulk (especially the write signature length). In contrast, LogTM-SE suggests that for workloads similar to SPLASH-2 [97], short signatures of 64 bits may be sufficient [98]. Since the STAMP applications spend more time in transactions and generate larger read and write sets than the SPLASH-2 benchmarks, it is likely that signature-based TM systems will need longer signatures for workloads like STAMP. This observation was confirmed in [81] for two of the STAMP applications on LogTM-SE.

Further experiments are necessary to determine if such signatures are sufficient for other applications and larger-scale systems. For workloads like `yada`, performance degrades significantly with shorter signatures on 32 cores. Fortunately, large scale systems are likely to use a directory instead of broadcast coherence. Using a directory filters bus traffic, which also makes it less likely for the false positives from shorter signatures to occur.

The choice of hash functions can also play a significant role in the signature accuracy. Detailed analyses of hash functions suitable for hardware signatures can be found in [22, 23, 74, 81]. In [82], Sanchez et al. also present an efficient hardware implementation of signatures as well as realistic area estimates based on commercial processors. They conclude that with parallel Bloom signatures of 4 Kbits and four hash functions, the AMD Barcelona processor [99] would have a core size increase of 0.25% and a die size increase of 0.10%. The corresponding data for the Sun Niagara processor [52] are 4.1% for the core and 1.1% for the die. Since SigTM has shorter signatures, its area overhead will be even smaller than these estimates.

Chapter 7

Conclusions

In this dissertation, I presented a workload-based design of an effective hybrid transactional memory system. First, I introduced the *Stanford Transactional Applications for Multi-Processing (STAMP)* to address the shortcomings of previous approaches used to evaluate TM systems. With STAMP, I analyzed different HTM and STM proposals to guide the design of *Signature-Accelerated TM (SigTM)*, my new hybrid transactional memory design that combines the advantages of HTM with those of STM. In this chapter, I will summarize both STAMP and SigTM and comment on some possible future work for each.

7.1 Stanford Transactional Applications for Multi-Processing

In creating STAMP, I sought to create the first benchmark suite for TM that adequately addressed *breadth* in application domains, *depth* in stressing a wide range of transactional execution cases, and *portability* across a wide range of TM systems. To demonstrate these properties, I ran 20 variants of the eight STAMP applications on six different TM systems: eager and lazy variants of HTM, STM, and hybrid TM. All the measured transactional characteristics ranged at least two orders of magnitude,

and although the TM systems' relative performance was oftentimes the expected behavior, STAMP helped identify cases that contradict conventional wisdom and require further research.

The source code for the STAMP applications and the lazy and eager STMs is freely available to the public from `http://stamp.stanford.edu`. My hope is that STAMP will help address a great need in the TM research community by providing a comprehensive tool for helping people design and evaluate TM systems. Already there are many early adopters of STAMP in both industry and academia [14, 25, 35, 45, 53, 82, 92].

In its current form, STAMP is a very useful tool for analyzing TM systems, but in the future, more benchmarks could be added to cover even more transactional scenarios. For example, a workload exhibiting frequent privatization and publication of objects could be added to test how well TM systems support strong isolation. Another possibly interesting case to target would be an application that exhibits various phases during its execution. For example, the workload could start off with long transactions with large read sets and low contention, move to short transactions with small write sets and high contention, and then end with long transactions with large write sets and low contention. Phased behavior such as this would help analyze how well TM systems can adapt to dynamic program behavior.

7.2 Signature-Accelerated Transactional Memory

The design goal of SigTM was to combine the performance characteristics and strong isolation guarantees of hardware TM techniques with the low cost and flexibility of software TM systems. To create the SigTM design, I first used STAMP to identify and analyze the bottlenecks of STM proposals. Based on these results, I designed SigTM to use hardware signatures to track the read set and write set for pending transactions, but implemented data versioning and all other transactional functionality in software. Unlike previous hybrid designs, SigTM requires no modifications to the hardware caches in a multi-core system, which reduces hardware cost and simplifies support for features such as nested transactions and multithreaded cores. SigTM is also the

first hybrid TM system that transparently provides strong isolation guarantees that lead to predictable interactions between transactional blocks and non-transactional accesses.

Using the STAMP applications, I compared SigTM to STM systems. I showed that SigTM outperforms software-only transactions by 50% to 100% on average and can perform up to 6× faster on some applications. I also demonstrated that for lazy SigTM, 1 Kbit read signatures and 128 bit write signatures are sufficient for eliminating most false conflicts due to the inexact nature of signature-based conflict detection. Eager SigTM requires a slightly longer write signature of 512 bits. Finally, for large scale systems, directories may be useful for reducing the number of false positives from short signatures.

Future work for SigTM can be grouped into two categories. First, enhancements to the SigTM design can be made to provide even greater acceleration of software transactions, thus reducing the performance gap between SigTM and HTM. For example, more hardware can be added to SigTM to implement a technique similar to HASTM's *mark bits* [79]. This would allow SigTM to eliminate the overhead of dynamically redundant read and write barriers. Another hardware improvement could be to dynamically change the signature hash functions depending on the exhibited application behavior. For example, a large number of aborts could trigger an adjustment of hash functions in hopes of remedying a pathological memory access pattern. The second area of future work for SigTM is to realize the design in an actual processor. For an industry looking to combine the advantages of HTM and STM, SigTM presents a promising solution.

Bibliography

[1] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, 2000.

[3] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Francisco, California, February 2005.

[4] D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *HiPC '05: 12th International Conference on High Performance Computing*, December 2005.

[5] W. Baek, C. Cao Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM transactional application programming interface. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.

[6] W. Baek, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. Towards soft optimization techniques for parallel cognitive applications. In *19th ACM Symposium on Parallelism in Algorithms and Architectures*. June 2007.

[7] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08,, Princeton University, 2008.

[9] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 1970.

[10] C. Blundell, J. Devietti, et al. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Computer Architecture News*, 35(2), 2007.

[11] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 24–34, New York, NY, USA, 2007. ACM.

[12] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[13] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), July–December 2006.

[14] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[15] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23–29, Jul-Aug 1999.

[16] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[17] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. June 2007.

[18] B. D. Carlstrom. *Programming with Transactional Memory*. PhD thesis, Stanford University, June 2008.

[19] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Cao Minh, L. Hammond, C. Kozyrakis, , and K. Olukotun. Executing Java programs with transactional memory. *Science of Computer Programming*, 63(10):111–129, December 2006.

[20] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Cao Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional Execution of Java Programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. University of Rochester, October 2005.

[21] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, New York, NY, USA, June 2006. ACM Press.

[22] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, New York, NY, USA, 1977. ACM.

[23] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[24] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 97–108, Washington, DC, USA, 2007. IEEE Computer Society.

[25] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 304–315, New York, NY, USA, 2008. ACM.

[26] D. M. Chickering, D. Heckerman, and C. Meek. A Bayesian approach to learning Bayesian networks with local structure. In *UAI '97: Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence*, pages 80–89.

[27] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, O. Colavin, and B. Calder. Unbounded page-based transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems.* ACM Press, October 2006.

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

[29] J. Chung, C. Cao Minh, A. McDonald, H. Chafi, B. D. Carlstrom, T. Skare, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM Press, October 2006.

[30] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the 12th International Conference on High-Performance Computer Architecture*, February 2006.

[31] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, October 2006.

[32] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC'06: Proceedings of the 20th International Symposium on Distributed Computing*, March 2006.

[33] D. Dice and N. Shavit. What really makes transactions faster? In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[34] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, March 2007.

[35] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[36] D. Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[37] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing*, pages 303–323. LNCS, Springer, Sep 2005.

[38] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 258–264, New York, NY, USA, 2005. ACM Press.

[39] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[40] B. Haagdorens, T. Vermeiren, and M. Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *WISA '04: Proceedings of the 5th International Workshop on Infomation Security Applications*, pages 188–203, 2004.

[41] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 102–113, June 2004.

[42] T. Harris. Exceptions and side-effects in atomic blocks. In *2004 PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[43] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[44] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[45] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 2008. ACM.

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

[47] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[48] 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. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 88–98, 11-15 Feb. 2006.

[49] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report.

[50] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous multi-threading implementation in POWER5. In *Conference Record of Hot Chips 15 Symposium*, Stanford, CA, August 2003.

[51] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[52] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March–April 2005.

[53] E. Koskinen and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA '08: Proceedings of the twentieth annual symposium on*

Parallelism in algorithms and architectures, pages 160–168, New York, NY, USA, 2008. ACM.

[54] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[55] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

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

[57] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, Sept. 1961.

[58] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *19th International Symposium on Distributed Computing*, September 2005.

[59] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[60] A. McDonald, B. D. Carlstrom, J. Chung, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Transactional memory: The hardware-software interface. *Micro's Top Picks, IEEE Micro*, 27(1), Jan/Feb 2007.

[61] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on*

Computer Architecture, pages 53–65, Washington, DC, USA, June 2006. IEEE Computer Society.

[62] 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: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–74, Washington, DC, USA, September 2005. IEEE Computer Society.

[63] C. McNairy and R. Bhatia. Montecito: The next product in the Itanium processor family. In *Conference Record of Hot Chips 16*, Stanford, CA, August 2004.

[64] A. W. Moore and M. S. Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, 1998.

[65] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *12th International Conference on High-Performance Computer Architecture*, February 2006.

[66] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. *Workload Characterization, 2006 IEEE International Symposium on*, pages 182–188, Oct. 2006.

[67] U. G. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. G. andAshok Kumar, and H. Park. An 8-core, 64-thread, 64-bit, power efficient sparc soc. In *Presentation at ISSCC 2007*, San Francisco, CA, February 2007.

[68] C. Perfumo, N. Sonmez, O. S. Unsal, A. Cristal, M. Valero, and T. Harris. Dissecting transactional executions in haskell. In *TRANSACT '07: Second ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, August 2007.

[69] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[70] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 5–17, New York, NY, USA, October 2002. ACM Press.

[71] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, June 2005. IEEE Computer Society.

[72] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: transactional memory for an operating system. *SIGARCH Computer Architecture News*, 35(2):92–103, 2007.

[73] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. Metatm/txlinux: transactional memory for an operating system. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 92–103, New York, NY, USA, 2007. ACM.

[74] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers*, 46(12):1378–1381, 1997.

[75] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, Sep 2006.

[76] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Jun 2007.

[77] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2007. ACM.

[78] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[79] B. Saha, A. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO '06: Proceedings of the International Symposium on Microarchitecture*, 2006.

[80] 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 eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

[81] D. Sanchez. Design and implementation of signatures for transactional memory systems. Technical Report CS-TR-2007-1611, Department of Computer Sciences, University of Wisconsin-Madison, August 2007.

[82] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society.

[83] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM Press.

[84] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Transactions and privatization in delaunay triangulation. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 336–337, New York, NY, USA, 2007. ACM.

[85] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, Canada, August 1995.

[86] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[87] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[88] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*. June 2007.

[89] M. F. Spear, V. J. Marathe, L. Dallesandro, and M. L. Scott. Privatization techniques for software transactional memory. Technical Report TR915, Computer Science Department, University of Rochester, February 2007.

[90] Standard Performance Evaluation Corporation, *SPECjbb2000 Benchmark*. `http://www.spec.org/jbb2000/`, 2000.

[91] Standard Performance Evaluation Corporation, *SPEC OpenMP Benchmark Suite*. http://www.spec.org/omp.

[92] J. R. Titos, M. E. Acacio, and J. M. Garcia. Characterization of conflicts in log-based transactional memory (logtm). In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 30–37, Washington, DC, USA, 2008. IEEE Computer Society.

[93] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *IEEE International Solid-State Circuits Conference (ISSCC 2008)*, San Francisco, CA, February 2008.

[94] E. Vallejo, T. Harris, A. Cristal, O. Unsal, and M. Valero. Hybrid transactional memory to accelerate safe lock-based transactions. In *TRANSACT '08: Third ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, February 2008.

[95] D. W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188. ACM Press, 1991.

[96] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 388–398, Washington, DC, USA, 2007. IEEE Computer Society.

[97] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[98] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[99] A. Zeichick. One, Two, Three, Four: A Sneak Peek Inside AMD's Forthcoming Quad-Core Processors. Technical report, AMD, January 2007.