ARCHITECTURES FOR TRANSACTIONAL MEMORY

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

Austen McDonald
June 2009

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.

(Oyekunle 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.

(Mendel Rosenblum)

Approved for the University Committee on Graduate Studies.

Abstract

Engineers have successfully worked for decades to improve single thread CPU performance, but we have now reached a peak in what a single thread can do. Average programmers are now facing the eventuality that their code must be parallel to take advantage of the performance potential of multi-core chips.

Unfortunately, writing parallel programs is hard because synchronizing accesses to shared state is complex and error-prone—many techniques have been tried, but achieving performance and correctness simultaneously still requires expert programmers, and the method of choice is decades old (locks). Transactional Memory (TM) is a relatively new programming paradigm promising an easier road to correctness and performance using atomic code regions. These regions may then be speculatively executed in parallel, potentially providing performance gains.

This dissertation focuses on architecting and evaluating hardware TM systems. We begin by briefly arguing that TM should be implemented in hardware, since proposed software solutions suffer from poor performance. We then study qualitatively and quantitatively the large design space for hardware TM as defined by primary options such as version management and conflict detection, and vary the secondary options such as the structure of the memory hierarchy, the instructions per cycle, and the configuration of the interconnect. Orthogonally, we determine the semantics and interfaces needed by any hardware TM system to support rich software functionality in modern operating systems and programming languages. Finally, we extend hardware support for transactional execution to create a multi-core architecture that provides cache coherence and memory consistency at the granularity of atomic transactions.

We conclude that programs written with transactional memory can achieve comparable to and often superior performance than the same programs written with traditional synchronization methods. Furthermore, a transactional architecture implementing lazy versioning and optimistic conflict detection is the preferred method of implementing TM in hardware due to its simplicity and good performance across a wide range of contention scenarios. Also, to support rich semantics, you need four mechanisms: two-phase transaction commit, software handlers, nested transactions, and non-transactional loads and stores. Finally, a continuously transactional architecture called Transactional Coherence and Consistency (TCC) maintains performance benefits while simplifying the hardware implementation of TM.

Acknowledgments

Special thanks to my advisor Christos, for picking me as his student and for guiding me through the long Ph.D. process, for teaching me how to do research, and for always reminding me of what's important "at the end of the day."

To Kunle, a.k.a., Papa K., a.k.a., the Wild Nigerian, thanks for keeping me on task and for being there to joke around with.

Also, a thank you goes out to Mendel Rosenblum, my third dissertation reader.

No acknowledgments page would be complete without thanking our hard-working administrators, Teresa Lynn and Darlene Hadding. And of course, thank you to Uncle Sam, the sources of the money they so skillfully administered: NSF Career Award 0546060, NSF Grants CNS-0720905 and CCF-0444470, FCRP contract 2003-CT-888, DARPA NBCH-104009, Army High Performance Computing Research Center HPTi W911NF-07-2-0027-1.

The work for this thesis was not completely my own, but is the product of numerous students' sweat and tears: to Brian David David Carlstrom, thanks for writing "The Robot," for managing the benchmarks, for the Java results, for numerous late-night debugging sessions, and for all the money you loaned me to go to the Thai Cafe; to JaeWoong Chung, even though you never learned to heed the coding standards, your numerous contributions to the simulator code were invaluable; to Hassan Chafi, for help debugging and for the color you added to our group; to Chi Cao Minh, thanks for parallelizing a bunch of

benchmarks and for creating STAMP; to Nathan Bronson, thanks for the EP simulator and for introducing me to climbing; to Ben Hertzberg, your x86 simulation code was greatly appreciated; and to Lance Hammond, for inspiring the simulator code and leaving me with the solemn duty of resident graph designer.

Thanks to my advisor at the Georgia Institute of Technology, Kenneth Mackenzie, and to his students, for teaching me what research is, for helping me get to Stanford, and for convincing me I should stay out of a hardware-based Ph.D.

There are too many friends to name, but thank you to all those who celebrated victories with me, persevered through my hard times, offered their shoulders to cry on, and gave me a life outside of the Gates Building.

Of course, I wouldn't be here without my loving. Thanks for threatening to come out here if I didn't finish and thanks especially for all your support throughout my entire life!

Finally, thanks be to God, to whom often I prayed the prayer of Moses: *May the favor of the Lord our God rest upon us; establish the work of our hands for us—yes, establish the work of our hands.* Psalm 90:17.

Contents

Abstract **v**

Acknowledgments **vii**

1 Introduction **1**

1.1 Transactional Memory 1

1.2 The Case for Hardware Transactional Memory 4

1.3 Thesis 5

1.4 Organization 6

2 The Architectures of HTM **7**

2.1 Basic TM Framework 8

2.1.1 Architectural Interface 9

2.1.2 Strong versus Weak Isolation 10

2.2 Eager-Pessimistic (EP) 11

2.3 Lazy-Optimistic (LO) 14

2.4 Lazy-Pessimistic (LP) 16

2.5 Eager-Optimistic (EO) 17

2.6 Contention Management 17

2.6.1 Contention Management Policies 18

2.6.2 Pathologies 19

2.6.3 Universal Contention Manager 21

2.7 Virtualization 23

2.8 Other Uses of HTM 25

2.9 Related Work 26

3 Evaluation of HTM Design Space **29**

3.1 Expected Performance 29
3.2 Experimental Setup 32
3.2.1 Eager-Pessimistic 33
3.2.2 Lazy-Optimistic 36
3.2.3 Lazy-Pessimistic 37
3.2.4 Contention Management 37
3.3 Benchmarks for Evaluation 39
3.3.1 bayes 39
3.3.2 genome 40
3.3.3 intruder 42
3.3.4 kmeans 42
3.3.5 labyrinth 42
3.3.6 ssca2 43
3.3.7 vacation 43
3.3.8 yada 44
3.3.9 barnes 44
3.3.10 mp3d 45
3.3.11 radix 45
3.3.12 swim 45
3.4 Baseline Evaluation 46
3.5 Contention Management in Pessimistic Conflict Detection 48
3.5.1 Discussion 51
3.6 Comparing Transactional Systems 52
3.7 Comparing to Traditional Parallelization 58
3.8 Shallow vs. Deep Memory Hierarchy 60
3.9 Instruction-Level Parallelism 63
3.10 Interconnect Parameters 66
3.11 Related Work 69

3.12 Conclusions 72

4 The Architectural Semantics of HTM 75

4.1 The Need for Rich HTM Semantics 77

4.2 HTM Instruction Set Architecture 79

4.2.1 Two-phase Commit 82

4.2.2 Commit Handlers 82

4.2.3 Violation Handlers 83

4.2.4 Abort Handlers 84

4.2.5 Nested Transactions 84

4.2.6 Nested Transactions and Handlers 86

4.2.7 Non-Transactional Loads and Stores 88

4.2.8 Discussion 89

4.3 Flexibly Building Languages and Systems 90

4.4 Hardware Implementation 93

4.4.1 Two-Phase Commit 93

4.4.2 Commit, Violation, and Abort Handlers 94

4.4.3 Nested Transactions 95

4.5 Evaluation 99

4.5.1 Performance Optimizations with Nesting 99

4.5.2 I/O within Transactions 101

4.5.3 Conditional Synchronization within Transactions 102

4.6 Conclusion 103

5 All Transactions All the Time 105

5.1 Continuous Transactional Execution 107

5.2 Transactional Coherence and Consistency 109

5.3 Design Alternatives 111

5.3.1 Coherence Granularity 111

5.3.2 Coherence Protocol 112

5.3.3 Commit Protocol 113

5.4 Performance Evaluation 114

5.4.1 Methodology 114
5.4.2 Baseline Evaluation 115
5.4.3 Coherence Granularity: Line-level versus Word-level 119
5.4.4 Coherence Protocol: Update versus Invalidate 121
5.4.5 Commit Protocol: Commit-through versus Commit-back 124
5.4.6 Bus Utilization 127
5.5 Conclusions 128

6 Conclusions and Future Work **131**

Bibliography **133**

List of Tables

2.1 The fundamental design space of transactional memory. 11
2.2 Qualitative performance evaluations in the fundamental TM design space. 12

3.1 Default simulator parameters. Experiments use this setup unless otherwise noted. 33
3.2 Cache and transactional statistics for benchmark applications. 41
3.3 Descriptions of each of the components of execution time. 46

4.1 State needed for rich HTM semantics. 80
4.2 Instructions needed for rich HTM semantics. 81
4.3 HTM mechanisms needed to implement various TM programming languages. 90

5.1 Cache and transactional statistics for benchmark applications under continuous transactions. 115
5.2 Speedups on each of the TCC systems. 130

List of Figures

2.1 Basic multicore TM environment. 8
2.2 How optimistic and pessimistic conflict detection schemes work. 13
2.3 Contention management pathologies. 20

3.1 Undo log unrolling implementation. 35
3.2 Execution time breakdown of STAMP on EP-BASE. 47
3.3 Execution time breakdown of STAMP on LO-BASE. 47
3.4 Execution time breakdown of STAMP on LP-BASE. 48
3.5 Comparing speedups of different the contention management configurations. 49
3.6 Execution time breakdown of the best evaluated contention management configurations, by application, of the first four STAMP applications across all systems. 54
3.7 Execution time breakdown of the best evaluated contention management configuration, by application, of the second four STAMP applications across all systems. 55
3.8 Execution time breakdown comparison between TM and traditional MESI parallelization. 59
3.9 Speedups of `kmeans` and `vacation` on LO, EP, and LP with and without a private L2. 62
3.10 Execution time breakdown of `vacation` on LO with only a private L1. . . . 62
3.11 Speedups of all three systems on a single issue, a 2-issue, and a 4-issue CPU. 63
3.12 Execution time breakdown, on all three systems, of selected STAMP applications using a higher arbitration time. 67

4.1 Timeline of three nested transactions: two closed-nested and one open-nested. 85
4.2 The Transaction Stack. 87
4.3 Conditional synchronization using open nesting and violation handlers . 91
4.4 Cache line structure for denoting nesting levels 96
4.5 Performance improvement with full nesting support over flattening. . . . 100
4.6 Performance of transactional conditional synchronization. 103

5.1 Execution time breakdown of STAMP on TCC-BASE. 117
5.2 Execution time breakdown of STAMP on TCC-WORD. 120
5.3 Execution time breakdown of STAMP on TCC-UPDATE. 122
5.4 Execution time breakdown of STAMP on TCC-BACK. 125
5.5 TCC bus utilization on STAMP applications. 128

Chapter 1

Introduction

Mainstream computing has reached a critical juncture. On one hand, with Instruction-Level Parallelism (ILP) techniques running out of steam, multicore processors are the norm. On the other hand, writing correct and efficient multithreaded programs with conventional programming models is still an incredibly complex task limited to a few expert programmers. Unless we develop programming models and execution environments that make parallel programming the common case, the performance potential of multicore machines will be limited to multiprogramming workloads and a few server applications.

1.1 Transactional Memory

A great deal of the difficulty in creating multithreaded programs comes from the need to manage shared state. When two threads need to communicate or operate on shared data, some mechanism must be employed to ensure correct, serializable execution. Mutual exclusion implemented via locks has been the traditional solution to this problem. Programmers use locks by surrounding accesses to shared state with `lock()` and `unlock()` pairs. Once a thread enters the lock region, no other threads are allowed to execute the region's code.

While simple in concept, locks present a number of challenges. First, locks present

programmers with a simplicity versus performance tradeoff. A program using few coarse-grained locks is easy to reason about and programmers have a good chance of generating a correct version of such a program. However, with large lock regions, performance will be poor, as mutual exclusion prevents parallelism. On the other hand, a program with many small locks is likely to perform better, but may be difficult implement correctly.

This difficulty stems from a number of sources. First, fine-grained locks require programmers to associate specific locks with specific data items. Unfortunately, this association is only in the programmer's mind, creating opportunities for careless bugs. Using monitors to incorporate locks into data structures [51] may alleviate this problem. However, monitors do not solve the problem of lock composability: using multiple locks requires strict programmer discipline to avoid deadlock and, in the case of monitors, is difficult to do without access to the locks themselves. For example, the published code for Sun's JDK 1.0.2 implementation of `java.lang.String.intern()` contains an atomicity bug because locks were not acquired in the correct order [15]. Additionally, pathological interactions, like priority inversion and convoying, can be created between locks and system code like context switches.

Another problem with locks is the poor debugging environment offered by most systems. Many mortal programmers who have written lock-based programs have struggled, at one time or another, with the above-listed problems of deadlock or simply forgetting to lock shared accesses. Unfortunately, finding these bugs is difficult without expensive profiling operations and tools. These tools are usually poor because, as we mentioned earlier, locks and the data they protect is not an easy relationship to establish automatically. Additionally, synchronization-related bugs often fail to reproduce consistently.

There are alternatives to locks like non-blocking synchronization, which avoids the deadlock and priority inversion problems of locks. These methods can offer better performance, but correctness is even harder: new problems are created like livelock and starvation. In fact, perfecting non-blocking algorithms even for simple data structures has been the subject of a number of Ph.D. theses.

To address these challenges with locks, it has been proposed that transactional techniques from the realm of databases [34] be applied to general-purpose programming

in the form of "transactional memory" (TM) [57, 48, 90]. In TM, large to medium-sized programmer-defined regions are executed by the runtime system atomically, automatically detecting conflicts. This is achieved by tracking what reads occur inside the transaction (the so-called read-set) and what writes occur (the so-called write-set). At some point before the transaction commits, its read- and write-sets are compared to other transactions to detect and resolve conflicts. In this way, transactions implement the A and I (atomicity and isolation) qualities of database transactions. Database transactions also implement consistency, but this is the responsibility of the TM programmer, who must properly managing his own data-structures, and durability, but TM operates on only memory objects and no persistence is guaranteed.

Transactions present a number of advantages over locks. First, by using one simple language construct, `atomic {}`, users no longer must manage named locks, allowing the compiler to check for closed regions. Additionally, transactions provide atomicity instead of mutual exclusion, which is usually what programmers want when they use locks. In this way, TM allows two transactions to speculatively execute in parallel, improving performance if they do not conflict. In this way, larger atomic regions can be used, and conflicts are detected only if they exist, instead of prevented via mutual exclusion.

Since `atomic` guarantees atomicity, it avoids the lock combining problem and because there are no named lock regions, composing `atomic` is trivial (to the programmer that is, we'll talk more about implementation of nested transactions in Section 4.4.3).

There are also enhanced benefits from TM, like easier debugging and using the TM constructs for other applications. Debugging tools are easier to implement because TM systems track information about program reads/writes and their interactions between threads. This information can then be exposed to the programmer and used to better tune and debug their program [20]. TM also has numerous applications outside of simply protecting shared memory accesses. The ability to roll back an action could be useful in security applications where operations should be tried before they are actually done. Also, a useful debugging technique called deterministic replay can be implemented using the rollback features of TM [68, 101, 28]. TM can be used to efficiently implement memory snapshots [28] which can be used for concurrent garbage collection, among other things.

In conclusion, transactional memory is a promising new parallel programming environment, empowering programmers to better utilize the newly available parallelism provided by multicore processors.

1.2 The Case for Hardware Transactional Memory

The first TM systems were hardware systems with very limited "transactions" [57, 48], but researchers soon extended the ideas from these proposals to what we know today as transactional memory [46, 41]. In fact, the fundamental mechanisms of TM may be implemented completely in hardware (we call these systems HTMs), completely in software (STMs), or in a mixture of the two (hybrid TMs). In this section we argue that HTM, even though requiring a more intensive development effort considering its hardware nature, is the preferred way to provide TM.

STMs do have a number of advantages over hardware systems. The biggest advantage of STMs is that they operate on existing hardware, providing immediate support for transactional memory. Also, as we discuss later (Section 2.7), HTM systems require extra support for transactions of arbitrary size and length, whereas STMs manage this case with ease. Additionally, since STMs can track read- and write-sets at very fine granularities, their conflict detection mechanisms may be less susceptible to false sharing. Another benefit of STMs has been the semantic flexibility: researchers have been able to experiment with various transactional languages and constructs without having to build hardware systems first.

Unfortunately, there are also a number of disadvantages to STMs. Read- and write-sets in STMs must be managed by added code, creating a great deal of overhead not present in HTMs. This overhead stands to significantly reduces performance gains from concurrency. Additionally, since this management code must be added to any code running inside a transaction, legacy and library code cannot be used within transactions without recompiling. Finally, if strong isolation is desired (see Section 2.1.2), this management code must be added to non-transactional code as well, further degrading performance. These disadvantages are unlikely to disappear unless programmers use very small, infrequent transactions, which mitigates the programming advantages of TM.

Because of these tradeoffs, researchers have proposed hybrid, hardware/software, solutions [14, 59, 31, 85]. While these systems begin to bridge the gap between STM and HTM performance, HTM is still significantly better [13].

STM advocates argue that quickly getting TM into the hands of programmers may help to convert them to TM, following up with hardware at a later date. But, first impressions are important: if a new TM programmer sees poor performance, he may not understand that hardware can make it faster, but may instead abandon TM all together. The remainder of this dissertation explores Hardware Transactional Memory: its implementations, performance, and design alternatives.

1.3 Thesis

This dissertation characterizes transactional architectures and explores the design points and potential alternatives. We compare performance between transactional architectures and traditional parallel architectures, examine the semantics needed to support real-world applications, and evaluate the performance and practical considerations surrounding continuous transactions.

My dissertation provides the following specific contributions:

- Programs written with transactional memory can achieve comparable performance to those written by experts with traditional synchronization methods.
- Furthermore, a transactional architecture called "Lazy-Optimistic" is the preferred method of implementing TM in hardware.
- Performance is not enough though, as TM must also support modern languages and operating systems. To do this, you need four mechanisms: two-phase transaction commit, software handlers, nested transactions, and non-transactional loads and stores.
- Finally, we can extend "Lazy-Optimistic" to use transactions as the only means of coherence and consistency, maintaining the performance benefits of TM while even further simplifying the hardware implementation.

1.4 Organization

The remainder of this dissertation is organized as follows.

Chapter 2 explores, in detail, the possible architectures for hardware transactional memory, including the so-called "Lazy-Optimistic" scheme, mentioned in the thesis above.

Chapter 3 evaluates the architectures described in Chapter 2 on a number of axes using transactional applications.

Chapter 4 describes the ISA-level semantics for and their implementation in a fully-functional transactional memory system that supports modern programming languages and operating systems.

Chapter 5 introduces the concept and semantics of continuous transactions, or "All Transactions All the Time," in a system called Transactional Coherence and Consistency (TCC). I then discuss the merits of such an approach, and evaluate the performance against non-continuous transactional systems.

Chapter 6 concludes, summarizing the impact of my work on the world of transactional memory and parallel computing.

Chapter 2

The Architectures of HTM

Fundamentally, there are three mechanisms needed to implement an atomic and isolated transactional region: *versioning*, *conflict detection*, and *contention management*. We will describe them briefly here and explain how to put them together to form complete systems in the following sections.

To make a transactional code region appear atomic, all its modifications must be stored and kept isolated from other transactions until commit time. The system does this by implementing a versioning policy. Two versioning paradigms exist: eager and lazy. An eager versioning system stores newly generated transactional values in place and stores previous memory values on the side, in what is called an undo-log. A lazy versioning system stores new values temporarily in what is called a writebuffer, copying them to memory only on commit. In either system, the cache is used to optimize storage of new versions.

To ensure serializability between transactions, conflicts must be detected and resolved. The system detects conflicts by implementing a conflict detection policy, either optimistic or pessimistic. An optimistic system executes transactions in parallel, checking for conflicts only when a transaction commits. Pessimistic systems check for conflicts at each load and store. Similar to versioning, conflict detection also uses the cache, marking each line as either part of the read-set, part of the write-set, or both. The system resolves conflicts by implementing a contention management policy. Many policies exist, some more appropriate for optimistic conflict detection and some more appropriate

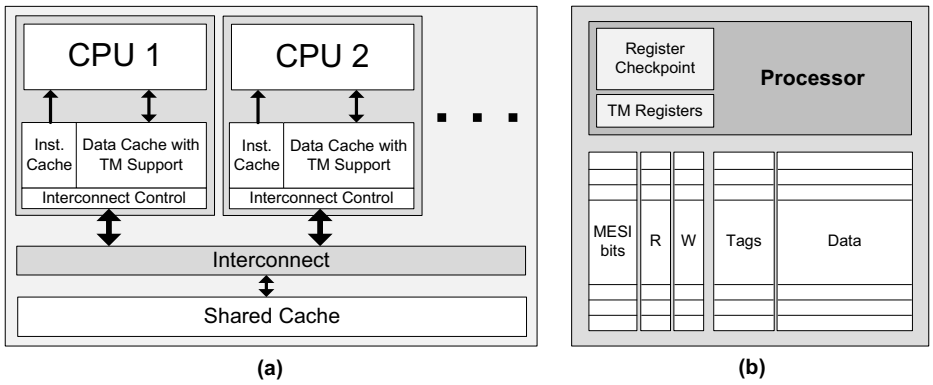

Figure 2.1: Our basic multicore TM environment. Part (a) shows many TM-enabled CPUs on one die, connected with an interconnect. Part (b) shows the details of a transactional CPU, including additions to support TM.

for pessimistic. In this chapter, we describe some popular policies and how they work.

Since each TM system needs both versioning and conflict detection, these options give rise to four distinct TM designs: Eager-Pessimistic (EP), Eager-Optimistic (EO), Lazy-Pessimistic (LP), and Lazy-Optimistic (LO). Table 2.1 briefly describes all four combinations and provides citations to the major proposed implementations of each design.

The remainder of this chapter describes the four basic designs and how to implement them, their advantages and disadvantages, and then discusses potential issues with TM implementations.

2.1 Basic TM Framework

For the sake of discussion, we describe TM architectures in the context of a multicore system similar to the one shown in Figure 2.1. Essentially, it is a number of small processor cores, each with some private cache, connected through an interconnect implementing the MESI coherence protocol or some derivative [5, 95]. The TM designs described in this chapter can be used on other architectures like traditional shared memory multiprocessors, distributed shared memory machines [69, 21], and probably others. We do not describe the changes required to implement TM on such systems.

To enforce versioning of registers, each core includes a fast, hardware register checkpointing mechanism, shown in Figure 2.1. These units are common in CPUs that support out-of-order execution and checkpoints [67]. Also shown are the additional bits in each cache line used to support conflict detection: a Read (R) bit and a Written (W) bit. Each system uses these to quickly test containment within the read- and write-set, respectively and we assume these can be quickly reset.

Other TM features include status registers (like one to determine whether a transaction is aborting or running) and configuration registers (like the ones required for contention management, see below). Each TM system proposed in literature describes a slightly different set and usage of such registers; a more complete description of the set we use can be found in Chapter 4.

2.1.1 Architectural Interface

All of our designs share a common architectural interface: they all share the same mechanisms to begin and end transactions. For the sake of discussion, let `tm_begin()` and `tm_end()` represent the code that begins and ends a transaction, respectively. These constructs can be used to build more advanced constructs like `atomic` [17]. `tm_begin()` consists of taking a register checkpoint and changing the status registers to reflect that a transaction has begun. It may also set up any undo log needed in an eager versioning system.

As further described in Chapter 4, `tm_end()` has two phases, called validation and commit by the database community [34]. The validation phase ensures that other transactions do not conflict with the completing transaction and commit makes that transaction's write-set available to other transactions. The available mechanisms for validation and commit are dictated by the choice of versioning and conflict detection policy.

Like commit, aborting a transaction depends on how versioning and conflict detection is performed, but there are a few commonalities. We split the abort process into two distinct operations (again, see Chapter 4 for motivation): `tm_discard()`, which discards the read- and write-set (R and W bits); and `tm_rollback()` which rolls the transaction back to its beginning, including restoring the register checkpoint.

2.1.2 Strong versus Weak Isolation

A key detail for programmers in any TM system is how non-transactional accesses interact with transactions. By definition, transactional accesses are screened from each other using the mechanisms above, but how will a regular, non-transactional load interact with a transaction containing a new value for that address? Or, how will a non-transactional store interact with a transaction that has read that address?

These are issues of the database concept isolation. A TM system is said to implement strong isolation (sometimes called strong atomicity in older literature) when every non-transactional load and store acts like an atomic transaction. Therefore, non-transactional loads cannot see uncommitted data and non-transactional stores cause atomicity violations in any transactions that have read that address. A system where this is not the case is said to implement weak isolation (sometimes called weak atomicity in older literature).

Strong isolation is desired because it is easier to reason about. Additionally, the programmer may have forgotten to surround some shared memory references with transactions, causing bugs. With strong isolation, the programmer will detect this using a simple debug interface because she will see a non-transactional region causing atomicity violations [20]. Also, programs written in one model may work differently on another model [11].

Strong isolation is easy to support in hardware TM: since the coherence protocol already manages load and store communication between processors, transactions can detect non-transactional loads and stores and act appropriately. To implement strong isolation in software TM, non-transactional code must be modified to include read- and write-barriers, potentially crippling performance. Great effort has been expended to remove un-needed barriers, but techniques are complex and performance still significantly worse than HTMs [91, 89].

Transactional Memory Design Space			
		VERSIONING	
CONFLICT DETECTION		*Lazy*	*Eager*
	Optimistic	Storing updates in a writebuffer; detecting conflicts at commit time. TCC [39], FlexTM [92], BulkTM [19]	Not practical: waiting to update memory until commit time but detecting conflicts at access time guarantees wasted work and provides no advantage.
	Pessimistic	Storing updates in a writebuffer; detecting conflicts at access time. LTM [8], VTM [80]	Updating memory, keeping old values in undo log; detecting conflicts at access time. LogTM [69], UTM [8], MetaTM [82]

Table 2.1: The fundamental design space of transactional memory (versioning and conflict detection). Included are references to significant recent proposals for HTMs of each type, if applicable.

2.2 Eager-Pessimistic (EP)

The first TM design we describe is Eager-Pessimistic. An EP system stores its write-set "in place" (hence the name "eager") and, to support rollback, stores the old values of overwritten lines in an "undo log". Processors use the W and R cache bits to track read- and write-sets and detect conflicts when receiving snooped load requests. Perhaps the most notable examples of EP systems in the literature are LogTM [69] and UTM [8].

Beginning a transaction in an EP system is much like beginning a transaction in other systems: `tm_begin()` takes a register checkpoint and initializes any status registers. An EP system also requires initialing the undo log, the details of which are dependent on the log format, but probably involves initializing a log base pointer to a region of pre-allocated, thread-private memory and clearing a log bounds register. Section 3.2 describes the log format we implemented for our experiments, but many other formats could be used.

Versioning: In EP, the MESI state transitions are left mostly unchanged due to the way eager versioning works. Of course, outside a transaction, MESI is completely unchanged. When reading a line inside a transaction, the standard coherence transitions apply (S→S,

Qualitative Comparison	
VERSIONING	
Lazy	*Eager*
+ No additional per store access penalty. − Commit may be slower, requiring per cache line memory transactions, making transactions with high writes to instructions ratios vulnerable to performance degradation. + Rollback is fast, with only local changes required.	− Per store access penalty: old value must be written to undo log. + Commit-in-place makes commit instantaneous. − Undo log must be applied making rollback slow.
CONFLICT DETECTION	
Optimistic	*Pessimistic*
+ Because conflicts are detected at commit time, no information need be exchanged on a per access basis. − At validation time however, the write-set must be communicated to all other transactions for conflict checking. − "Doomed" transactions, containing access that will cause conflicts are allowed to continue, wasting resources. + However, more serializable schedules are allowed.	− Conflict detection on the critical path: each load must be seen by all transactions to detect conflicts and a contention management decision may potentially be made on each access. + No additional validation time required since exclusive rights to lines are acquired on access. + "Doomed" transactions die early. − But, not all serializable schedules are allowed.

Table 2.2: Qualitative performance evaluations in the fundamental TM design space.

I→S, or I→E), issuing a load miss as needed, but the R bit is also set. Likewise, writing a line applies the standard transitions (S→M, E→I, I→M), issuing a miss as needed, but also sets the W bit. The first time a line is written, the old version of the entire line is loaded then written to the undo log to preserve it in case the current transaction aborts. The newly written data is then stored "in-place," over the old data.

Conflict Detection: Pessimistic conflict detection uses coherence messages exchanged on misses or upgrades to look for conflicts between transactions (see Figure 2.2(a)). When a read miss occurs within a transaction, other processors receive a load request. Of course, they ignore the request if they do not have the line. If they have it non-speculatively

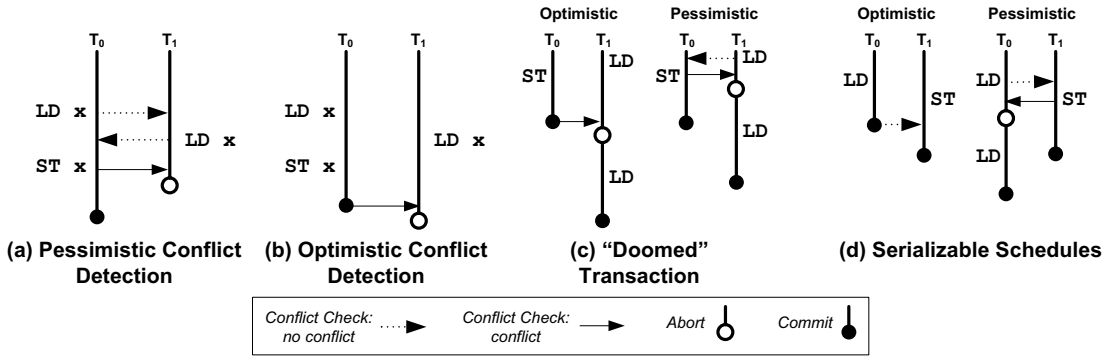

Figure 2.2: How optimistic and pessimistic conflict detection schemes work and some tradeoffs in conflict detection. (a) pessimistic conflict detection. (b) optimistic conflict detection. (c) how optimistic conflict detection wastes work by allowing "doomed" transactions to continue, while pessimistic does not. (d) how optimistic allows more serializable schedules to coexist, while pessimistic does not.

or have the line R, they downgrade their line to S and perhaps issue a cache-to-cache transfer if they have the line in MESI's M or E state. But if the cache has the line W, then a conflict is detected between the two transactions and some action must be taken immediately.

Similarly, when a transaction seeks to upgrade a line from shared to modified (on its first write), it issues an exclusive load request, which is also used to detect conflicts. If a receiving cache has the line non-speculatively, they invalidate it and perhaps issue a cache-to-cache transfer (M or E states). But, if the line is R or W, a conflict is detected.

Validation: Because conflict detection is performed on every load, a transaction always has exclusive access to its write-set. Therefore, validation does not require any additional work.

Commit: Since eager versioning stores the new version of data items in place, the commit process simply clears the W and R bits and discards the undo log (this is very fast).

Abort: When a transaction rolls back, the original version of each cache line in the undo log must be restored, a process called "unrolling" or "applying" the log. This is done during `tm_discard()` and must be atomic with regard to other transactions. Specifically, the write-set must still be used to detect conflicts: this transaction has the only correct

version of lines in its undo log and requesting transactions must wait for the correct version to be restored from the log. The log can be applied using a hardware state machine or software abort handler. For our systems, we implemented a software log unroller, described in Section 3.2.

Advantages: Commit is simple and since it is in-place, very fast. Similarly, validation is a no-op.

Pessimistic detects conflicts early, as shown in Figure 2.2(c), avoiding "doomed" transactions: T_0 and T_1 are involved in a Write-After-Read dependency which is detected immediately in pessimistic conflict detection, but not until the writer commits in optimistic.

Disadvantages: As described above, the first time a cache line is written, the old value must be written to the log, incurring extra cache accesses. Aborts are expensive as they require undoing the log. For each cache line in the log, a load must be issued, perhaps going as far as main memory before continuing to the next line.

Pessimistic conflict detection also prevents certain serializable schedules from existing. Figure 2.2(d) describes this phenomenon: T_0 reads a line and T_1 later writes to the same line. When T_1 attempts to upgrade its line to M, it detects a conflict and aborts T_0. But, a serializable schedule exists for these transactions (namely T_0, T_1), and is allowed by optimistic conflict detection.

Additionally, because conflicts are handled as they occur, there is a potential for livelock and careful contention management mechanisms must be employed to guarantee forward progress (see Section 2.6).

2.3 Lazy-Optimistic (LO)

Another popular TM design is Lazy-Optimistic (LO), which stores its write-set in a "write-buffer" or "redo log" and detects conflicts at commit time (still using the R and W bits). An LO system exploits different tradeoffs than an EP system, and has been chiefly advocated by the TCC [39] system, further described in Chapter 5.

Versioning: Just as in the EP system, the MESI protocol is enforced outside of transactions. Once inside a transaction, reading a line incurs the standard MESI transitions

but also sets the R bit. Likewise, writing a line sets its W bit, but handling the MESI transitions is different. First, with lazy versioning, the new versions of written data are stored in the cache hierarchy until commit while other transactions have access to old versions available in memory or other caches (see potential problems with cache overflow in Section 2.7). To make available the old versions, dirty lines must be evicted when first written by a transaction. Second, no upgrade misses are needed because of optimistic conflict detection: if a transaction has a line in the S state, it can simply write to it and upgrade to M without communicating with other transactions because conflict detection is done at commit time.

Conflict Detection and Validation: To validate a transaction and detect conflicts, LO communicates the addresses of speculatively modified lines to other transactions only when it is preparing to commit (see Figure 2.2(b)). On validation, the processor sends one, potentially large, network packet containing all the addresses in the write-set. Data is not sent, but left in the committer's cache and marked dirty (M). To build this packet without searching the cache for lines marked W, we use a simple bit vector, called a "store buffer," with one bit per cache line to track these speculatively modified lines.

Other transactions use this address packet to detect conflicts: if an address is found in the cache and the R and/or W bits are set, a conflict is initiated. If the line is found but neither R nor W is set, the line is simply invalidated, like processing an exclusive load.

Of course, to support transaction atomicity, these address packets must be handled atomically, i.e., no two address packets may exist at once with the same addresses. In our LO system, we achieve this by simply acquiring a global commit token before sending the address packet (see Section 3.2). However, a two-phase commit scheme could be employed by first sending out the address packet, collecting responses, enforcing an ordering protocol (perhaps oldest transaction first), and committing once all responses are satisfactory [21].

Commit: Once validation has occurred, commit needs no special treatment: simply clear W and R bits and the store buffer. The transaction's writes are already marked dirty in the cache and other caches' copies of these lines have been invalidated via the address packet. Other processors can then access the committed data through the regular coherence protocol.

Abort: Rollback is equally easy: because the write-set is contained within the local caches, we simply invalidate these lines then clear W and R bits and the store buffer. The store buffer allows us to easily find W lines to invalidate without the need to search the cache.

Advantages: Aborts are very fast, requiring no additional loads or stores and making only local changes. More serializable schedules can exist (see Figure 2.2(d) and the discussion for EP), which allows an LO system to more aggressively speculate that transactions are independent, perhaps improving performance. Finally, late detection of conflicts can make guaranteeing forward progress easier (see Section 2.6).

Disadvantages: Validation takes global communication time proportional to size of write-set. Doomed transactions can waste work since conflicts are detected only at commit time. See Figure 2.2(c) and the above description for EP.

2.4 Lazy-Pessimistic (LP)

Lazy-Pessimistic (LP) represents a third TM design option, sitting somewhere between EP and LO: storing newly written lines in a writebuffer but detecting conflicts on a per-access basis. LP has been proposed in the form the LTM [8] and VTM [80] systems.

Versioning: Versioning is similar but not identical to that of LO: reading a line sets its R bit, writing a line sets its W bit, and a store buffer is used to track W lines in the cache. Also, dirty lines must be evicted when first written by a transaction, just as in LO. However, since conflict detection is pessimistic, load exclusives must be performed when upgrading a transactional line from I,S→M, unlike in LO.

Conflict Detection: LP's conflict detection operates the same as EP's: using coherence messages to look for conflicts between transactions.

Validation: Like in EP, pessimistic conflict detection ensures that at any point, a running transaction has no conflicts with any other running transaction, so validation is a no-op.

Commit: Commit needs no special treatment: simply clear W and R bits and the store buffer, like in LO.

Abort: Rollback is also like that of LO: simply invalidate the write-set using the store buffer and clear the W and R bits and the store buffer.

Advantages: Like LO, aborts are very fast. Like EP, its pessimistic conflict detection avoids doomed transactions (Figure 2.2(c)).

Disadvantages: Like EP, some serializable schedules are not allowed (Figure 2.2(d)) and conflict detection must be performed on each cache miss.

2.5 Eager-Optimistic (EO)

The final combination of versioning and conflict detection is Eager-Optimistic (EO). Unfortunately, EO is not a logical choice for HTM systems: since new transactional versions are written in-place, other transactions have no choice but to notice conflicts as they occur (i.e., as cache misses occur). But since EO waits until commit time to detect conflicts, those transactions become "zombies," continuing to execute, wasting resources, yet are doomed to abort. We do not implement or evaluate any EO system.

EO has proven to be useful in STMs and is implemented by Bartok-STM [43] and McRT [86]. A lazy versioning STM needs to check its writebuffer on each read to ensure that it is reading the most recent value. Since the writebuffer is not a hardware structure, this is expensive, hence the preference for write-in-place eager versioning. Additionally, since checking for conflicts is also expensive in an STM, optimistic conflict detection offers the advantage of performing this operation in bulk.

2.6 Contention Management

We've learned how a transaction rolls back once the system has decided to abort it but, since a conflict involves two transactions, which one should abort, how should that abort be initiated, and when the aborted transaction should be retried? These are questions of Contention Management (CM), a key component to transactional memory. In this section, we describe how the systems we implemented initiate aborts and the various established methods of managing which transactions should abort in a conflict (called the Contention Management Policy).

2.6.1 Contention Management Policies

A Contention Management (CM) Policy is a mechanism to determine which transaction involved in a conflict should abort and when it should be retried. For example, it is often the case that simply retrying immediately does not lead to the best performance, but employing some backoff mechanism would be better. STMs first grappled with finding the best contention management policies and many of the policies outlined below were originally developed for STMs.

Policies draw on a number of measures to make their decisions, including ages of the transactions, size of read- and write-sets, the number of previous aborts, etc. Combinations are endless, but certain combinations have been used in the literature and we describe a number of them here, roughly in order of increasing complexity.

To establish some nomenclature, first note that in a conflict there are two sides: the attacker and the defender. The attacker is the transaction requesting access to a shared memory location. In pessimistic conflict detection, the attacker is the transaction issuing the load or load exclusive. In optimistic, the attacker is the transaction attempting to validate. The defender is the transaction receiving the attacker's request.

The Aggressive [47] policy immediately and always retries either the attacker or the defender. In LO, Aggressive means that the attacker always wins, and so Aggressive is sometimes called committer wins. It was used for the earliest LO systems [66]. For EP, aggressive can be either defender wins or attacker wins.

Restarting a conflicting transaction that will immediately experience another conflict is bound to waste work—namely interconnect bandwidth refilling cache misses. The Polite [47] policy employs exponential backoff (but linear could also be used) before restarting conflicts. To prevent starvation, it guarantees transaction success after some n retries.

Of course, one could just choose to randomly abort the attacker or defender (a policy called Randomized [87]). The inventors combine this with a randomized backoff scheme to avoid unneeded contention.

Perhaps wiser than making random choices is to avoid aborting transactions that have done "a lot of work." One measure of work could be a transactions age. Oldest is a

simple timestamp method that aborts the younger transaction in a conflict [79]. BulkTM uses this scheme [19]. SizeMatters [82] is like Oldest but instead of transaction age, the number of read/written words is used as the priority. The inventors revert to Oldest after a fixed number of aborts. Karma [87] is similar, using the size of the write-set as priority. Rollback then proceeds after backing off a fixed amount of time. Aborted transactions keep their priorities after being aborted (hence the name Karma). Polka [88] works like Karma but instead of backing off a predefined amount of time, it backs off exponentially more each time.

Since aborting wastes work, it is logical to argue that stalling an attacker until the defender has finished their transaction would lead to better performance. Unfortunately, such a simple scheme easily leads to deadlock: T_0 reads X, T_1 reads Y, T_0 tries to write Y, stalls on T_1, while T_1 tries to write X, and stalls on T_0.

Deadlock avoidance techniques can be used to solve this problem. Greedy [36] uses two rules to avoid deadlock: First, if T_1 has lower priority than T_0, or if T_1 is waiting for another transaction, then T_1 aborts when conflicting with T_0. Second, If T_1 has higher priority than T_0 and is not waiting, then T_0 waits until T_1 commits, aborts, or starts waiting (in which case the first rule is applied). Greedy provides some guarantees about time bounds for executing a set of transactions. One EP design (LogTM [69]) used a CM policy similar to Greedy to achieve stalling with conservative deadlock avoidance.

We did not cover every contention management policy devised (for example, Kindergarten and Eruption [88]), but we provided a picture of the CM design space. In Chapter 3 we evaluate a few of these policies.

2.6.2 Pathologies

Unfortunately, choosing a CM policy is difficult, not only because there are so many available policies, but also because predicting performance using a particular policy is difficult. Frequently encountered transaction mixes often conspire to create *pathologies*, which are contention phenomena that reduce performance. Some pathologies simply degrade performance while others create worse problems like starvation or livelock. This section describes some of these pathologies (illustrated in Figure 2.3) and what can

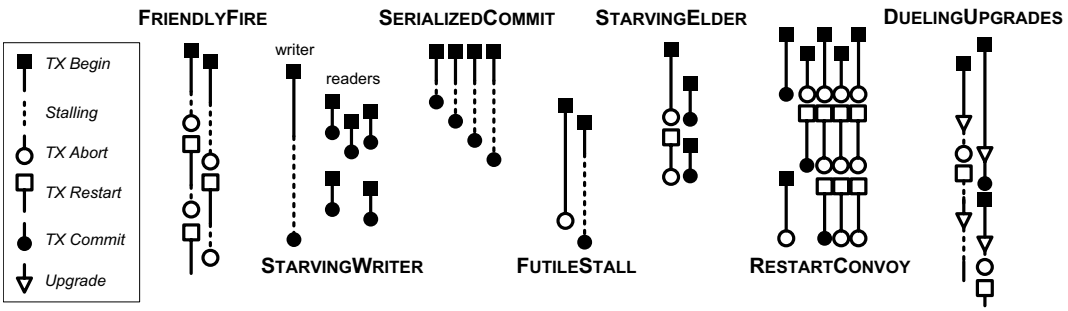

Figure 2.3: Illustration of the contention management pathologies. Reproduced from Bobba et al. [12].

be done to alleviate them. Pathologies were first categorized by Bobba et al. [12], and their work contains more details.

The FRIENDLYFIRE pathology may arise when pessimistic conflict detection is combined with an attacker-wins Aggressive CM policy. When one transaction conflicts with and aborts another transaction, then subsequently is aborted by a third transaction. This, of course, can lead to livelock but can be mitigated by using randomized linear backoff before restarting.

STARVINGWRITER may occur in a pessimistic system when one transaction attempts to write and conflicts with a number of concurrent readers. If a CM policy is used that stalls attackers without combining with a priority scheme to ensure forward progress, the writer may stall on the readers, but before the writer can retry its accesses, another reader has appeared. Using a policy like Stall (described in Section 3.2), which combines stalling with transaction age, will guarantee forward progress but performance may still suffer.

FUTILESTALL may also occur in a pessimistic system with a stalling policy when one transaction stalls waiting for another transaction that eventually aborts. In this case, stalling just wasted time. This can be exacerbated in EP because aborting takes extra time to undo the log.

Another pathology that occurs in EP systems with a stalling policy is DUELINGUPGRADES: when two transactions both read then attempt to write (i.e., upgrade) the same address, conflicts are detected and they attempt to stall on each other, with the deadlock avoidance protocol eventually aborting one. If the continuing thread commits then

begins another similar transaction, which frequently occurs in loops, then the restarted transaction may conflict with the new transaction, entering the pathology pattern once again.

Two pathologies may occur in LO systems with an Aggressive, attacker-wins CM policy. Since small transactions reach their commit phase before longer transactions, small transactions can starve older, longer ones, creating the STARVINGELDER pathology. Some priority scheme is required to prevent starvation in these cases. If many similar, conflicting transactions execute concurrently, then the first to commit may abort all others, creating a so-called RESTARTCONVOY. Members of the convoy may repeatedly execute, wasting system resources even though all but one are doomed to abort. A backoff mechanism can help to alleviate RESTARTCONVOY.

SERIALIZEDCOMMIT can occur in optimistic conflict detection systems that serialize at commit time to achieve a global order. Many small transactions may attempt to commit simultaneously, creating contention for global commit permission even though there may be no conflicts. This can be fixed by using a parallel commit scheme like that one described in the LO section above.

In conclusion, it is important to choose a proper contention management policy that avoids pathologies, but this is highly dependent on application characteristics. Pessimistic conflict detection leads to more and more likely pathologies than optimistic, though empirical studies (like those in Chapter 3) are required to determine if pathological transaction mixes really do exist and what their performance impact may be.

2.6.3 Universal Contention Manager

To implement these contention management policies, we introduce a mixed hardware and software solution called the Universal Contention Manager (UCM). The goals of the UCM are to make rapid decisions about which transaction should abort, so as not to degrade performance on each access, but also provide enough flexibility to support many CM policies.

In any policy, the actual abort process begins when one of these transactions executes a software violation handler (see Section 4.2.3 for details). The minimum required

handler would call `tm_discard`, `tm_rollback`, and perhaps undoing the log if the system is EP. This violation handler is the software component of the UCM and is key to supporting a wide range of CM policies without needed to design specific hardware structures to implement each one. Specifically, software handlers have access to any hardware registers tracking transaction state (like length and number of lines read/written) and can store their own software state.

To decide which processor will invoke its software violation handler, the UCM uses two global hardware registers (SENSE and TIEBREAK) and an additional per-processor register (PRIORITY). The PRIORITY register is sent over the interconnect along with each item of coherence traffic (e.g., load miss).

The SENSE register determines which transaction will run its handler: the one with the smaller priority register (SENSE=SMALLERWINS), the one with the larger priority register (SENSE=BIGGERWINS), or to defer to the TIEBREAK setting (SENSE=ALWAYSTIE).

The TIEBREAK register specifies which transaction, the defender or the attacker, should execute their handler in the case of a tie on the priority register or if SENSE is ALWAYSTIE. Possible values for TIEBREAK are DEFENDERWINS (i.e., run the handler on the attacker) or ATTACKERWINS (i.e., run the handler on the defender).

Some examples will help the reader understand the UCM. To implement Aggressive, we do not use the priority register, so we set SENSE to ALWAYSTIE and TIEBREAK to ATTACKERWINS to ensure that the committing transaction is allowed to proceed. The software violation handler then simply executes `tm_discard()` and `tm_rollback()`.

To incorporate the transactions' ages using the UCM (e.g., to implement Oldest), we could set the priority register to the time during `tm_begin()`. Oldest would then set SENSE to SMALLERWINS and TIEBREAK to either DEFENDER or ATTACKERWINS. Similarly, to incorporate abort counts or write-set sizes, the software violation handler can store these values in thread-private variables, perhaps copying them into the PRIORITY register for future rapid comparison to the values from other transactions.

Different abort characteristics can also be implemented using the software violation handler. Linear and exponential backoff can be implemented by simply spinning in a loop before calling `tm_discard(); tm_rollback()`.

The UCM is very simple hardware yet achieves important goals: making abort conflict resolution decisions quickly and locally while maintaining the flexibility of a software handler.

2.7 Virtualization

Because transactions use caches to track read- and write-sets, a number of issues can arise when cache space is exhausted or the cache is needed for another thread, like on a context switch. These are issues of so-called transactional virtualization. This section describes some of the challenges and related work.

First, restricting the user to transactions whose metadata (W and R bits) and data fit in local caches is unacceptable. Studies have found that the common-case behavior of transactional systems is small transactions [27], but also that reasonable programming practices result in transactions too large for some cache structures [8], inducing what is called "cache overflow." Certain architectural extensions can be employed, like victim caches, to significantly reduce the probability that such transactions will exist [66], but no matter how large are local caches, systems will need some method of virtualizing space.

Second, if transactions are employed on a large scale, then it is certain that context switches, page faults, process migration, etc. will occur during a transaction. How can transactional semantics be maintained when cache structures can no longer be used exclusively for one transaction? This problem is called time virtualization. Major challenges include maintaining isolation of yet-to-be-committed data and maintaining conflict detection between running transactions and those "swapped out" of the caches.

There are a number of goals for these virtualization systems. Obviously, they must support transactional semantics like atomicity and isolation, but the presence of virtualizing mechanisms (i.e., hardware and software overhead) should also not affect the common-case transactional performance. Additionally, once a virtualized transaction exists, it is preferable that the performance of other transactions in the system not suffer because of it. Finally, since we assume virtualization will be rarely used, additional complexity (especially hardware complexity) should be avoided.

There have been a number of proposals to virtualize transactions, not all achieving every goal or even virtualizing both time and space. What follows is a brief look at some of the proposed mechanisms.

One simple approach to handle space virtualization, chosen by us in early research on LO systems [66], is simply to acquire a global token preventing others from validating, flush out the transaction's write-set (essentially committing), then continue execution, holding the token until encountering a user-defined commit. This approach guarantees atomicity of the overflowed transaction, but punishes the entire system, not just the overflowed transaction. It is also important to note that this approach exposes a transaction's writes to the system before it commits, preventing that transaction from aborting on its own and violating strong isolation (giving non-transaction threads access to uncommitted state). We continue to employ this approach to simplify implementation of our LO system because overflows are rare in our applications.

Ananian et al. [8] were the first to propose an HTM design with more significant solutions to virtualization problems, namely their UTM system, which is EP. UTM is an idealized system utilizing additional per-location memory pointers and global virtual addresses to virtualize both time and space, but at significant overheads. The same work introduces LTM, a more reasonable system, but it lacks time virtualization and limits space virtualization to the size of main memory.

Like LTM, LogTM [69] virtualizes space but not time. One advantage of an eager versioning system like LogTM is that the undo log can grow to arbitrary size without special handling. However, metadata overflow still occurs and LogTM chooses to simply keep the W/R bits on overflowed lines, OR-ing any future metadata modifications. Of course, this causes additional conflicts but it is a simple low-overhead solution. Later LogTM designs used signatures for conflict detection, allowing time virtualization by saving/restoring signatures [105].

VTM [80] is an LP system supporting time and space virtualization. It supports two execution modes, one for common case, non-virtualized transactions, and one for virtualized transactions. Virtualized transactions have their overflowed write-set mapped to virtual memory and similarly, swapped-out transactions also store their state in virtual memory. To accomplish this, VTM uses an additional hardware-managed hash table

and a Bloom filter to perform rapid conflict checks.

A number of software-enhanced virtualization mechanisms have also been proposed. HybridTM [31] simply switches to software transactions when virtualization is needed, but requires two copies of all transactional code: one for the HTM and one for the STM. XTM [26] and Page-based TM [23] (PTM) both use extra pages to enable space virtualization. XTM employs lazy versioning, buffering writes in a newly created page, while PTM employs eager versioning, storing its writes in place, keeping the last committed version in a new page. Also, XTM is mostly software while PTM employs more hardware structures. Enhancements to XTM also support line-granularity virtualization.

In conclusion, implementing time and space virtualization is important in any real TM system, but it should be thought of as a rarely-used feature. Virtualization systems need to be tailored to the chosen versioning and conflict detection mechanisms. Eager versioning may make space virtualization easier since you do not have to store new versions separately. Lazy versioning may make virtualizing time easier since you do not have to protect other loads/stores from eagerly-committed data. Despite these biases, efficient solutions exist at all TM design points.

2.8 Other Uses of HTM

The hardware TM employs for supporting atomicity and optimistic concurrency can also be used to accelerate a number of important systems applications, not just multi-threaded applications. Fundamentally, TM provides isolated memory regions and highly-observable application behavior, which have wide-ranging applicability in OSs, debugging, optimization, reliability, and security.

Conflict detection requires constant program introspection. This can be used to provide debugging and optimization support. With Transactional Application Profiling Environment (TAPE) [20], we suggested using a conflict logging system to identify potentially fruitful optimizations and to detect non-transactional accesses to shared state. Similarly, having hardware track read- and write-sets enables data race detection [63, 74, 75].

By providing isolation, HTMs already have support for a kind of hardware memory

snapshot, which can greatly improve the ease of developing a host of often serialized application components including concurrent garbage collectors, dynamic profilers, and copy-on-write, as observed by Chung et al. [28].

ASeD [25] uses the mechanisms provided by TM as building blocks for Availability, Security, and Debugging. The authors show how versioning hardware can be used to support recovery schemes to deal with hardware failures, both permanent and transient, and to provide isolation for suspicious code. They also show how TM-supported fine-grained address protection can be used to implement canaries and low-overhead read/write barriers for security. Finally, ASeD supports hardware watchpoints and the ability to step backward through multi-threaded code.

Other work has also been done to exploit TM's features to build deterministically replayable multi-threaded systems [68, 52, 32, 75]. We discuss this possibility more in Chapter 5.

Finally, as an easier-to-implement synchronization mechanism, TM can be used in environments where generating correct multi-threaded code is complex. For example, shared access to the metadata used to perform Dynamic Binary Translation can be made thread-safe using TM [24], which enables many applications like runtime optimization, debugging, and security analysis.

2.9 Related Work

Major proposed systems that represent each of the TM design points have been cited above. For a more thorough discussion of TM developments as of 2006 see Larus and Rajwar [61]. This section describes some additional TM systems and other related work.

Sun's Rock [33] is a recent highly-speculative multicore processor with a isolating hardware checkpointing feature. Rock uses this mechanism to implement a "best effort" HTM scheme designed to accelerate software transactions without concern for properly handling all events that may occur within a transaction (like TLB misses, page faults, etc.). They find that Rock's HTM can improve performance and make a number of suggestions on using and improving it.

MetaTM [82] is an EP system the designers used to evaluate a transactional version

of Linux, including evaluating a number of contention managers using software abort handlers. An extension to MetaTM, DATM [81] introduces dependence-aware transactions, altering the coherence protocol to forward dependencies between transactions instead of aborting them, significantly improving performance.

A number of authors have advocated decoupling versioning and conflict detection to build flexible TM systems [50, 92]. They argue that architecting a TM system in this way is good because it fosters early adoption of transactional memory without locking designers into one system. Just as we discussed in the previous section, versioning and conflict detection can also be used to implement a number of other useful features.

Recent TM systems have been proposed that replace the use of cache bits for conflict detection with aggregated address signatures, usually implemented using Bloom filters [19, 105]. This should reduce hardware complexity in the latency-sensitive local caches and eases virtualization, especially of time (see Section 2.7). Of course, false conflicts may arise, degrading performance.

Chapter 3

Evaluation of HTM Design Space

The HTM design space described in Chapter 2 embodies a number of design decisions. In this section we analyze the performance implications of those decisions, attempting to answer a number of questions.

First, what are the performance implications of choosing one TM design approach (Lazy-Optimistic, Eager-Pessimistic, or Lazy-Pessimistic) over another and how do they perform under different workloads and contention management policies? What happens to performance when we vary parameters like depth of the memory hierarchy, available instruction-level parallelism, and interconnect parameters? Second, is HTM's performance comparable to that of traditional shared memory multiprocessors? If the programmability advantages of TM come at a large performance cost, then TM is unlikely to be practical. Finally, given what we've learned from these experiments, what is the best recommended TM design and why?

3.1 Expected Performance

Before measuring the performance using quantitative methods, it is useful to discuss our expectations from a qualitative point of view. This section provides an initial assessment of the three different HTM designs and the expected answers to the questions posed in the introduction.

TM Designs, Workloads, and Contention Management

Fundamentally, the three TM systems all provide the same programming interface. However, each system embodies implementation tradeoffs in versioning and conflict detection summarized in Table 2.2. Because of these differences, we cannot expect performance to be the same between systems under all circumstances. For example, if commit is the common case for most transactions, perhaps lazy's bulk commit operations will degrade performance compared to eager's commit-in-place. On the other hand, eager's log writes create more cache pressure, also perhaps degrading performance. Only quantitative experiments will determine if and when either of these effects are significant.

We expect performance differences to grow with contention. If restarted transactions contain many writes, then the eager versioning system will incur significant overhead undoing the log, whereas lazy systems rollback quickly. On the other hand, optimistic systems may waste work in "doomed transactions," waiting for conflicts to be detected once transactions commit. A pessimistic system avoids this wasted work by detecting conflicts early, but may abort a transaction even though it is part of a serializable schedule.

It is these effects that create the contention management pathologies described in Section 2.6.2. Many pathologies have been identified in pessimistic systems and the best solutions are difficult to predict without experimentation. On the other hand, optimistic's abort-related pathologies (STARVINGELDER and RESTARTCONVOY) are few and require very specific transaction access patterns, leading us to believe they will be rare in practice. Our contention management experiments will help us decide how frequent are pathologies and how difficult it is to mitigate them in practice.

Traditional Parallel Architectures

TM may provide an easier-to-use alternative to lock-based synchronization, but if comparable speedups cannot be achieved with TM, programmers may abandon it. Comparing the two systems on the same benchmarks should show how much, if any, performance is sacrificed for a simplified programming environment.

We expect any performance differences to arise from differences between lock overhead and transaction overhead. Highly-turned lock-based applications have very small

lock regions and if these applications are converted to transactions by changing `lock` and `unlock` to `tm_begin` and `tm_end`, respectively, the resulting transactions will also be small. Beginning and ending transactions require more overhead (about 20 cycles to begin, assuming cache hits, and 4 cycles to end, assuming instant commit) than beginning and ending lock regions (9 cycles to acquire, 4 cycles to release), especially with software handlers like those we implement described in Chapter 4, so applications converted in this manner should experience some performance degradation compared to lock-based synchronization. On the other hand, if lock regions are large, potentially creating mutual exclusion where none is needed, then TM's optimistic concurrency may result in better performance for transactions.

Shallow Memory Hierarchy

TM systems use local cache space to store speculative state, turning to overflow mechanisms when this space is exhausted. Without evaluating the effectiveness or performance of different overflow mechanisms, what can we say about the impact of reducing local cache space?

Obviously, we should see decreased performance in applications with large working sets, due to capacity misses. But how many overflows will be seen in the workloads we have chosen to evaluate? If overflows become common, we expect our implementations of LO and LP, when used with simple serializing overflow mechanisms, to perform poorly. Our EP implementation however, should perform well since its overflowed transactions do not serialize the system.

Instruction-Level Parallelism

Eager versioning requires two extra cache accesses for each store incurring a log write—one to read the old value and one to write it to the log—for a total of three accesses. Ideally, these would be hidden by subsequent non-memory instructions, thereby avoiding processor stall. However, as instructions per cycle (IPC) increases, it may become difficult to hide these extra accesses without an additional L1 data port.

Our baseline comparison uses an in-order, single issue CPU design. To understand the impact of IPC on EP's log writes, we evaluate the benchmarks on a system with greater issue width. At higher IPCs, we expect the single-ported data cache to become

a bottleneck and for EP's performance to suffer relative to the other systems. The magnitude of performance degradation will depend on the instruction mix within transactions, specifically the ratio of write-set size to transactional instructions.

Interconnect Parameters

Because systems use the shared interconnect differently, we expect they will be affected differently by varying the parameters of this interconnect. For example, increasing the time to acquire permission to send data (arbitration time) will adversely affect all systems but LO may especially suffer because of its extra bus transactions to commit. EP and LP may also suffer because they acquire exclusive access to each line they write before the transaction proceeds.

Just as altering interconnect latency interacts with TM overheads, so does altering available bandwidth. Of course, decreasing bandwidth should slow all systems, but since write-set addresses are broadcast at commit time in LO, its performance may suffer acutely. Similarly, if any log writes are spilling out of the private caches, EP will experience additional slowdown.

3.2 Experimental Setup

To evaluate the HTM systems, we built an execution driven simulator based on Lance Hammond's extensions [37] to SimOS [84]. It models a number of single-issue, in-order x86 cores (compliments of Ben Hertzberg's x86 frontend [49]) in a CMP environment, with a full memory hierarchy. The CMP organization we study is similar to those in previous studies [10, 38, 58]: a number of simple processors with private L1s sharing an additional large, on-chip shared cache. Our system also has an exclusive private L2, like some of AMD and Intel's recent designs [5, 95]. Each private cache has a fully associative, 16-entry victim cache. The processors and shared cache are connected through split-transaction request and refill buses. The two buses provide high bandwidth through wide data transfers and bursts. The request bus has de-multiplexed address and data lines and is used to initiate shared accesses (address only) and writebacks (address and data). The refill bus is used to transmit refill data and cache-to-cache transfers (data

CPU	1–32 in order, single-threaded, single-issue x86 cores
L1	64KB, 32B cache line, 2-way associative, 1 cycle latency, 16 entry victim cache
L2	Private, exclusive 512KB, 32B cache line, 16-way associative, 3 cycle latency, 16 entry victim cache
Bus Width	32 bytes
Bus Arbitration	3 pipelined cycles
Bus Transfer Latency	3 pipelined cycles
Shared L3 Cache	8MB, 16-way, 8 banks, 20 cycles hit time
Main Memory	100 cycles latency, up to 8 outstanding transfers

Table 3.1: Default simulator parameters. Experiments use this setup unless otherwise noted.

only). Both buses are logical but not physical buses, providing serialization and broadcast. To support high bandwidth, they are implemented using a star-like topology with point-to-point connections that supports pipelined operation for both arbitration and transfers [56].

For all systems, we used single-ported data caches with dual-ported tags. This allows us to perform loads and process snoop requests in the same cycle, but if a cache-to-cache transfer is needed or if a log write is being processed, we cannot use the cache for loads/stores from the processor.

The default configurations are described in Table 3.1. We will vary those parameters throughout this chapter to explore different aspects of HTMs; the differences will be noted appropriately.

There are a number of low level decisions made in architecting any one of the three basic HTM designs. The following sections describe these decisions for each design.

3.2.1 Eager-Pessimistic

Building an efficient undo log key to obtaining reasonable performance in an eager versioning system. Our log design consists of a number of groups, contiguous in memory, each group beginning with one cache line full of addresses—our 32B cache lines hold 8 of these addresses. Following this group header are the 8 line-sized and line-aligned data items holding the old values at the addresses in the header.

To build this log during execution, the processor buffers speculatively written cache line addresses until a full header is collected, then writing it as one, line-sized store. Data log entries are written immediately to the log, using a single store, when a non-transactional line is first written in a transaction. This means the cache is unavailable for three subsequent cycles: one to read the old line, one to write the old line to the log, and one to write the new word coming from the processor.

Care must also be taken when designing a mechanism to undo the log. Some systems have implemented this in hardware [69]. Using our log structure, it is conceivable that a hardware undo mechanism could take as little as 4 cycles assuming cache hits: one to read the address, one to read the old data, one to write the old data, and one to increment the address counter. While this is undoubtedly more efficient than a software undo handler, we feel our software solution is quite efficient and easily enables a variety of contention management policies (see Section 4.2.3 about violation handlers).

Figure 3.1 presents a high-level language version of our software log unroller. To increase its efficiency, we introduced a special instruction, `LINE_MOVE`, which copies an entire cache line of data from one address to another. This is done in 2 cycles, assuming cache hits. Without this special instruction, or some other vector instruction like it, our system would incur an additional 8 loads and 8 stores (one for each word), or 16 cycles, per line. The inner loop of our software handlers is compiled and unrolled to a 10 instruction kernel, which executes in 11 cycles (assuming cache hits). In other words, undoing a cache line takes an amortized 11 cycles.

As described in Section 2.2, atomic abort requires continuing to detect conflicts on the transaction's write-set until the log unroller has restored the original versions of speculatively written lines. In our system, when a conflict is detected with a currently aborting transaction, the requester will not abort, which would waste work, but rather retry its request immediately until its desired line is available.

To handle overflow, our eager system employs a Bloom filter approach similar to LogTM-SE [105]: the addresses of speculative evicted lines are registered with a per-processor Bloom filter. This filter is then checked when processing snoop requests, thereby maintaining conflict detection without global performance loss. We keep one Bloom filter for overflowed, speculatively read lines and one for overflowed, speculatively written

```
const WORD_SIZE = 4
const LINE_SIZE = 32
const WORDS_PER_LINE = LINE_SIZE / WORD_SIZE

// log entries are grouped with one line full of
// addresses and then a set of data lines
const GROUP_SIZE = (1 + WORDS_PER_LINE) * LINE_SIZE
const STRIDE = GROUP_SIZE / WORD_SIZE

lh = GET_LOG_HEADER()

// the outer loop walks groups of addresses
for (group = lh.lastGroup;
     group >= lh.firstGroup AND group != 0;
     group -= STRIDE)
{
   // the inner loop walks data lines within groups
   for (i = WORDS_PER_LINE - 1; i >= 0; --i) {
      // first line in group contains addresses...
      addr = group[i]
      if (addr == NULL) {
        continue
      }
      // ...all subsequent lines contain data
      data = group[(1 + i) * WORDS_PER_LINE]

      LINE_MOVE(data, addr)
   }
}
```

Figure 3.1: Implementation of undo log unrolling in a high level language. Inner loop is compiled into a 10 instruction, unrolled kernel.

lines, each 2,048 bits wide. There is a small chance of false positive snoop matches on the Bloom filter, causing additional aborts, but this is rare.

3.2.2 Lazy-Optimistic

Only a few notes need to be made about the details of lazy's implementation. As mentioned in Section 2.3, if only one copy of each line can exist in the cache, lazy versioning systems must write back dirty lines before transactionally writing to them. We found that this significantly increases the amount of bus traffic. To alleviate this bottleneck, we utilized the hardware support for nesting levels described in Section 4.4.3 to store the original dirty line at nesting level 0 and copy it into nesting level 1 for use in a transaction. In this way, a speculative write to a dirty line does not immediately incur a writeback.

We implement validation and commit slightly differently than described in Section 2.3. To validate its transaction, a processor acquires a global commit token, preventing other transactions from validating. Commit then sends out the packet containing write-set addresses. The execution time breakdowns in this section's graphs reflect this division of work between validation and commit. Obviously, an application with many small transactions may experience significant contention for this commit token. We chose this implementation for simplicity, but the original description in Section 2.3 describes a more efficient parallel commit scheme.

We implement overflow in LO by serializing the overflowed transaction using the global commit token. First, the overflowed transaction acquires the token then "commits," sending a commit packet and marking its write-set as committed, allowing more speculative state to be stored in the cache. The token is not released and no other transactions may commit until the overflowed transaction completes. Unfortunately, this causes performance degradation to the entire system. There are many better virtualization options (discussed in Section 2.7), but since overflows were rare in our applications, we did not implement or evaluate them.

We use a bit vector to model a full-sized store buffer (up to the size of the private caches), able to designate each cache line as being speculatively written or not.

3.2.3 Lazy-Pessimistic

LP's versioning implementation is similar to LO's, including using the hardware support for nesting levels to avoid writing back dirty lines and the design of the store buffer to invalidate lines on abort. Validation and commit, of course, are different since pessimistic conflict detection precludes the need for explicitly communicating the write-set at commit time.

Overflow is also different, being a kind of hybrid between LO and EP's: LP's overflow does not require an explicit "commit" step as does LO's, but instead lines are simply evicted when needed. Conflict detection is then maintained by using a Bloom filter, just as in EP. Since the overflowed transaction cannot recall the writes it has evicted to memory, it cannot abort. Therefore, the first step in LP's overflow mechanism is to acquire a global token. This serializes overflowed transactions but allows other non-conflicting transactions to commit, since LP's transactions do not require a token to commit. If any transactions conflict with the overflowed transaction, once it has acquired the token, those transactions must abort.

3.2.4 Contention Management

For optimistic conflict detection, we chose the Aggressive contention manager. See Section 2.6.1 for a description and the UCM settings. We call LO with Aggressive LO-BASE.

It's interesting to note that the simple contention management policy for optimistic conflict detection provides for guaranteed forward progress (but not fairness). This makes LO attractive from a simplicity perspective. Unfortunately, it takes more effort to provide a similar guarantee for pessimistic conflict detection.

We also used Aggressive as the default policy for our pessimistic systems. Specifically, we chose the "defender wins" variant, where a requesting transaction aborts upon discovering a conflict with another transaction. We use this contention manager for the EP-BASE and LP-BASE configurations.

Unfortunately, "defender wins" does not guarantee forward progress. One of many possible livelock patterns starts with T0 writing X, T1 writing Y. Then T0 attempts to read Y but aborts, seeing that T1 already has rights to Y. T1 then attempts to read X but aborts

seeing T0 already has rights to X. This pattern then repeats, with no threads making progress.

Later in our investigation, we also evaluate two more CM policies for pessimistic conflict detection, in an attempt to find livelock and deadlock-free policies that also improve performance. We evaluate Oldest, where the younger transaction involved in a conflict aborts. This guarantees that, at all times, at least the oldest transaction makes forward progress. We call our systems EP-OLDEST and LP-OLDEST that use this policy.

In pessimistic conflict detection, aborts are expensive, not only because they waste work already done, but also because unrolling the undo log is expensive. Since avoiding aborts is advantageous, it could be wise to stall a requesting transaction that encounters a conflict, wait until the conflicting transaction commits or aborts, then continue with the request. This is the idea behind the Stall CM policy, which is similar to baseline policy of a popular EP design, LogTM [69] and also similar to Greedy. LogTM implemented their policy in hardware, but it can easily be implemented using the UCM. Each thread keeps, in software, a `possibleDeadlock` flag, set in the software contention manager when a younger transaction is waiting on it. We then use the UCM settings `SENSE: ALWAYSTIE`, `TIEBREAK: DEFENDERWINS`, and set the priority register to the time at the beginning of the transaction. When an attacker sees a conflict, it stalls unless its `possibleDeadlock` flag is set and the defender is older (checking the priority register to determine this), in which case it aborts.

Stall calls for waiting for the conflicting transaction to commit or abort, which requires global knowledge of commit and abort events. We did not implement a system that includes this knowledge as it would require even more changes to the coherence protocol, but instead stall by waiting for 500 cycles then retrying the request, perhaps stalling again if a conflicting transaction still exists. We adjusted this number but we do not claim to have found the optimal value. Too small a stall creates too many bus requests, and too large a stall wastes time.

Many other CM policies exist for pessimistic conflict detection (see Section 2.6.1) but we did not implement them. Our goal was not to present a thorough study of contention management policies, but rather to create a set of competitive TM architectures for comparison. Some discussion of preliminary results with other policies is in Section 3.5.1.

3.3 Benchmarks for Evaluation

To evaluate the HTM systems described above, we use the benchmarks in STAMP [13] version 0.9.10, and in our comparison with traditional parallelization methods, we selected benchmarks from SPEC CPUFP [97], SPLASH [94], and SPLASH-2 [104]. Table 3.2 summarizes measured application characteristics.

STAMP allows us to evaluate transactional systems with applications traditionally considered difficult to parallelize. The parallel code for each application uses coarse-grain transactions to execute concurrent tasks that operate on one or more irregular data structures such as graphs or trees. The resulting code is very close to the sequential algorithm, creating frequent, coarse-grain transactions. The characteristics of STAMP's transactions vary widely, making TM's performance on STAMP a good measure of the effectiveness of this new programming model.

On the other hand, SPEC, SPLASH and SPLASH-2 allow us to compare transactional systems using applications that represent important workloads amenable to traditional synchronization methods. Programmers have exerted great effort optimizing the performance of these benchmarks, so they present a worst-case evaluation of the additional overhead of TM's easier programming model.

From SPEC CFP we chose `swim`, from SPLASH we chose `mp3d`, and from SPLASH-2 we chose `barnes` and `radix`. We parallelized these applications by simply replacing the lock regions with transactions. This made for very small transactions as can be seen in Table 3.2 and is probably not representative of how a TM programmer would parallelize these applications directly from sequential versions. However, it is still instructive to evaluate their performance. It should be noted that simply replacing locks with transactions may create correctness issues, especially in the presence of benign races [11].

The remainder of this section describes each benchmark in detail.

3.3.1 bayes

`bayes` implements an algorithm for learning the structure of Bayesian networks from observed data, which is an important part of machine learning. The network is represented by a directed, acyclic graph with nodes for variables and edges for dependencies.

On each iteration, a thread analyzes a variable using the existing dependencies and the observed data to learn and add new dependencies. Since most of `bayes`'s time is spent analyzing the dependency graph and since a transaction protects these accesses, `bayes` has long transactions. `bayes` also has a high contention rate as newly discovered dependencies often change the graph.

The algorithm uses a hill climbing approach, combining local and global search, to converge to a solution. Because of this randomizing effect, we expect convergence rates to heavily depend on transaction scheduling. Long transactions and high contention may result in widely varying transaction schedules between TM systems, therefore producing equally varying execution times. To combat this effect, we average the performance from three runs of `bayes`, each with different random seeds, for all experiments involving this application.

Input: Dependencies for 32 variables are learned from 1,204 records, which have 2×20 parents per variable on average. Edge insertion has a penalty of 2, and up to 2 edges are learned per variable.

3.3.2 genome

`genome` is a bioinformatics application that performs gene sequencing: from a large pool of gene segments, it finds the most likely original sequence. The basic data structure is a hash table for unmatched segments. In the parallel version of the segment matching phase, each thread tries to add to its partition of currently matched segments by searching the shared pool of unmatched segments. Since multiple threads may try to grab the same segment, transactions are used to ensure atomicity.

Little contention exists in `genome`, despite execution time being spent mostly in transactions, and transactions are small. Because of these factors, we expect `genome` to perform similarly across TM systems.

Input: Gene segments of 16 nucleotides are sampled from a gene with 256 nucleotides. A total of 16,384 segments are analyzed to reconstruct the original gene.

	GLOBAL L2 MISS		LOCAL L3 MISS			PER TRANSACTION								
		per 1K ins		per 1K ins	Time in Tx	instructions*		reads*†		writes*†		writes† per 1K ins*		retries‡
bayes	0.22%	1.5	58.59%	1.24	83.3%	60,578	(614)	149	(43)	97	(18)	1.6	(29.3)	0.70
genome	0.41%	1.8	86.37%	1.59	97.4%	1,717	(241)	48	(36)	12	(14)	7.1	(58.1)	0.10
intruder	0.18%	0.9	79.37%	0.73	32.7%	330	(82)	21	(10)	10	(7)	31.6	(85.4)	2.08
kmeans	0.25%	0.9	37.10%	0.35	6.6%	118	(143)	12	(14)	5	(5)	42.6	(35.0)	0.07
labyrinth	0.01%	0.1	99.33%	0.05	99.7%	219,572	(194)	132	(23)	217	(12)	1.0	(61.9)	0.71
ssca2	3.44%	19.0	20.72%	4.95	15.2%	50	(50)	10	(10)	4	(4)	80.0	(80.0)	0.01
vacation	1.52%	9.6	33.76%	3.35	86.6%	2,612	(2,778)	96	(97)	20	(21)	7.7	(7.6)	0.41
yada	0.29%	2.0	51.57%	1.63	99.8%	9,801	(121)	51	(13)	28	(8)	2.9	(66.1)	0.52
barnes	0.01%	0.1	56.13%	0.08	1.4%	227	(62)	12	(9)	10	(6)	42.7	(96.8)	0.07
mp3d	0.09%	0.5	43.16%	0.22	61.0%	71	(56)	6	(5)	4	(4)	61.1	(71.4)	0.04
radix	0.27%	1.9	24.13%	0.67	0.0%	38	(38)	5	(5)	3	(3)	78.9	(78.9)	0.00
swim	5.55%	44.5	60.66%	34.23	0.0%	45	(45)	7	(7)	5	(5)	111.1	(111.1)	0.00

Table 3.2: Cache and transactional statistics for benchmark applications. *Averages are presented, with medians in parentheses. †Reads and writes are numbers of cache lines (32B) read and written, not number of read and write operations. Simulation parameters are the defaults presented in Table 3.1. ‡Retries per transaction presented for 16 CPU, Lazy-Optimistic case.

3.3.3 intruder

`intruder` is a signature-based network intrusion detection simulation where network packets are compared in parallel to known intrusion signatures. Packets are reassembled (presumably after being fragmented in transmission) using a shared dictionary, implemented using a self-balancing tree. Access to this tree is protected by coarse-grain transactions creating a high contention rate, so we expect performance to differ between systems. Also, it has a high writes to transactional instructions ratio, which may expose log and commit overheads.

Input: 2,048 flows, each consisting of not more than 4 packets. 10% of the flows are attacks.

3.3.4 kmeans

`kmeans` is an algorithm that clusters objects into k partitions based on some attributes. It is commonly used in data mining workloads. Input objects are partitioned across threads and synchronization is necessary when two threads attempt to insert objects in the same partition. Thus, the amount of contention varies with the value of k. Our version of `kmeans` is commonly called `kmeans-high` or `kmeans-high-vios` in other literature.

Despite being the "high contention" edition of `kmeans`, our measured contention level is relatively low compared to other applications. Therefore, we expect performance to be similar between systems. However, it does have many small transactions, so there is a possibility that LO will experience overhead from its serialized commit.

Input: High Contention Edition: The number of cluster centers is 15. A convergence threshold of 0.05 is used, and analysis is performed on an input with 2,048 points of 16 dimensions generated about 16 centers.

3.3.5 labyrinth

`labyrinth` is a three dimensional maze solving algorithm where threads choose a start and end point, then finds a path connecting the points through the grid and add their path to a global maze structure. This is done within one transaction. While computing

the path, each thread has its own privatized copy of the grid, verifying its path against the global structure once a successful path is found by re-reading all its edges. Transactions are ideal for this application because the checking phase would require a global lock or some other deadlock avoidance/detection technique.

High contention and long transactions may make this application sensitive to transaction scheduling. Spending a lot of time in long transactions as `labyrinth` does may pose problems for our optimistic system since it waits to detect conflicts until commit time.

Input: The reference input with a 32×32×3 grid, routing 96 paths.

3.3.6 ssca2

`ssca2` is an implementation of Scalable Synthetic Compact Applications 2 [9], which applies four commonly used kernels to a large, directed, weighted multi-graph. The STAMP version of this application focuses on Kernel 1, which constructs an efficient graph using adjacency and auxiliary arrays. Transactions protect access to the adjacency array and adding nodes to the graph, which are relatively small components of execution time. This makes transactions few and contention low so we expect systems to perform similarly. However, small transactions mean possible validation and/or commit overhead in LO.

Input: There are 2^{13} nodes in the graph. The probability of inter-clique edges and unidirectional edges are 1.0 and 1.0, respectively. The max path length is 3, and the max number of parallel edges is 3.

3.3.7 vacation

`vacation` implements a travel reservation system powered by an in-memory database using trees to track items, orders, and customer data. `vacation` is similar in design to the SPECjbb2000 [98] benchmark. The workload consists of several client threads interacting with the database via the system's task manager. The workload generator can be configured to produce a certain percentage of read-only (e.g., ticket lookups) and read-write (e.g., reservations) tasks. For our experiments, we used one with a balanced

set of read-only and read-write tasks to generate significant contention. Our version of `vacation` is also called `vacation-high` or `vacation-high-vios` in other literature.

Tasks operate on multiple trees and execute fully within transactions to maintain the database's atomicity, consistency, and isolation. `vacation` has a medium to high contention rate so performance differences between designs may emerge.

Input: High Contention Edition: 4 queries per task, 60% of relations queried, 16,384 possible relations, 90% user tasks, 4,096 tasks.

3.3.8 yada

`yada` (Yet Another Delaunay Application) implements Delaunay mesh generation, an algorithm for producing guaranteed quality meshes for applications such as graphics rendering and PDE solvers. The basic data structure is a graph that stores mesh data. Each parallel task involves three transactions. The first one removes a "bad" triangle from a work queue, the second processes the cavity around the triangle, and the third inserts newly created triangles into the work queue.

Since most of `yada`'s work is done within transactions and since it continually modifies a shared graph, its contention rate is high and we expect different performance characteristics on different TM systems.

Input: 633.node test file with 1,264 elements. Mesh must have a minimum angle of 20 degrees.

3.3.9 barnes

`barnes` is a SPLASH-2 benchmark simulating the interactions of a number of bodies (like planets for example) in three dimensions using the Barnes-Hut N-body method. Its principle data structure is an octree that is traversed on each time step, for each body, to compute the forces acting on that body.

`barnes` has the most contention of the traditionally parallelized applications we examined, as locks/transactions protect access to nodes in the tree. We expect to see conflicts play a key role in determining performance, perhaps helping us understand the differences between how traditional systems and TM systems handle contention.

Input: Reference input but with only 2,048 particles.

3.3.10 mp3d

`mp3d` is a SPLASH benchmark simulating rarefied fluid flow. It was excluded from SPLASH-2 because of its poor scalability, which makes it an ideal candidate to for parallelization using transactions, since it proved cumbersome with traditional methods. It has many lock regions and moderate contention (compared to other SPLASH applications), so it will test the performance limits of short transactions.

Input: 3,000 molecules.

3.3.11 radix

In `radix`, an iterative algorithm sorts a number of integer keys using a radix sort method. On each iteration, a thread generates a histogram of the keys it has been assigned, which are then accumulated into a global histogram. Finally, each thread uses the global histogram to permute its assigned keys.

`radix` uses no locks (except those in barriers), but instead has its own user-defined synchronization (i.e., "done" flags). Even though Table 3.2 lists `radix` as having 0% time in transactions, this is a roundoff effect: the only transactions are within barriers, so there are very few.

Input: 262,144 keys.

3.3.12 swim

`swim` comes from the SPEC CFP95 benchmarks and solves shallow water equations using finite difference methods. We parallelized it using standard techniques. `swim` has only one, short global reduction phase that requires synchronization, that is why it appears from Table 3.2 that 0% of time is spent in transactions. Like `radix`, `swim` should scale well on any parallel system, including transactions, and tests only basic synchronization pressure.

Input: Reference input.

Useful	Cycles spent executing non-memory instructions that contributed to the final result. Includes overhead from `tm_begin()` and `tm_end()` (but not time from any associated misses).
Overflow	Cycles of all types when a transaction has overflowed.
L1 Miss	Cycles spent accessing the private L2.
Memory	Cycles spent waiting for interconnect, shared cache, and main memory (i.e., time after the last private cache).
Idle/Synch	Cycles spent in barriers, locks, and idle (from load imbalance).
Validate	Cycles spent validating the transactional read-set.
Commit	Cycles spent committing.
Violate Stall	Rolled back cycles that were memory or cache stalls.
Violate	All remaining cycles that were rolled back.
Total	Total cycles.

Table 3.3: Descriptions of each of the components of execution time. "Total" does not appear in all graphs.

3.4 Baseline Evaluation

We start our evaluation by examining the results from the default configurations of the three systems described in Chapter 2 (LO-BASE, EP-BASE, and LP-BASE). Figures 3.3, 3.2, and 3.4 present the results as breakdowns of execution time, normalized to sequential execution (lower is better). Table 3.3 describes each component of the breakdown graphs.

Clearly, pessimistic's simple contention management scheme creates many livelocks, especially on applications with high contention like `bayes`, `intruder`, `vacation`, and `yada`. Here, frequent conflicts conspire to create crushing pathologies as many threads attempt to gain rights to a small number of cache lines.

Pessimistic's plight is unfortunate but expected since its CM scheme does not guarantee forward progress. Fortunately, LO's equally simple scheme does guarantee forward progress, avoiding livelock and, furthermore, appears not to create pathologies, except SERIALIZEDCOMMIT in `ssca2` (high Validation time on 32 CPUs due to small transactions). Other optimistic pathologies like STARVINGELDER or RESTARTCONVOY do not appear. We expected this outcome, as those pathologies require very specific thread configurations.

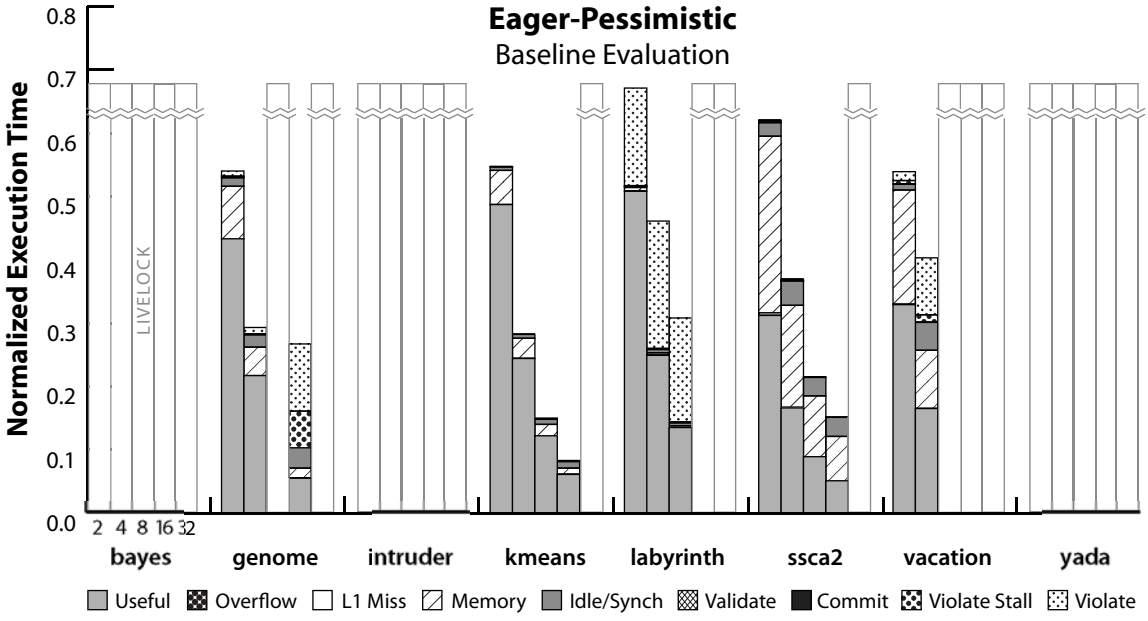

Figure 3.2: Execution time breakdown of STAMP applications on 2–32 CPUs on EP-BASE. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

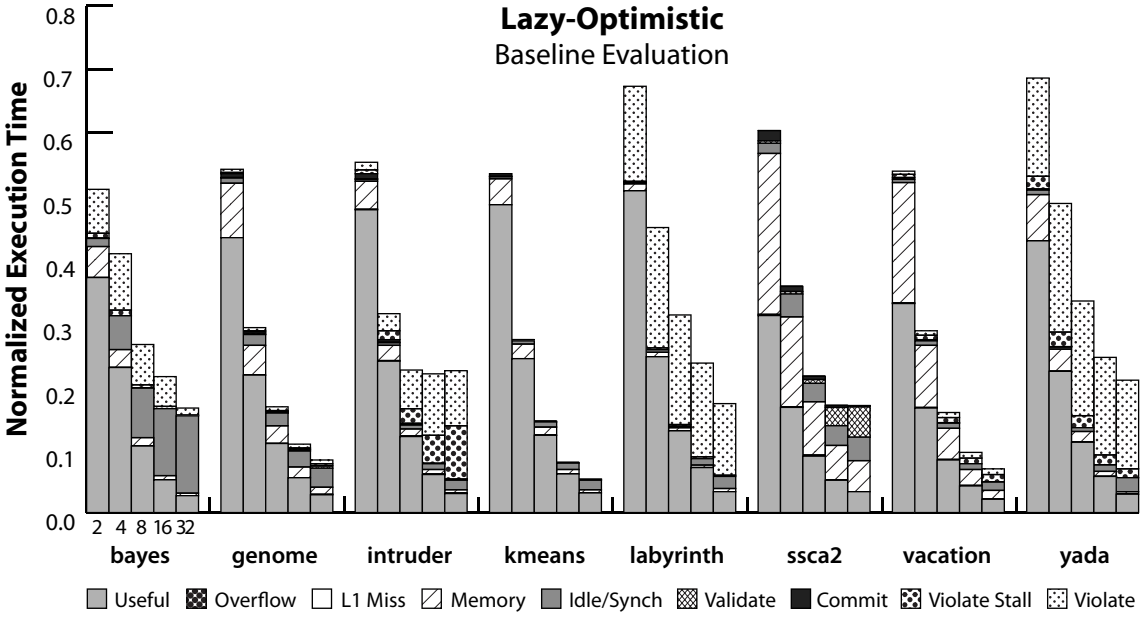

Figure 3.3: Execution time breakdown of STAMP applications on 2–32 CPUs on LO-BASE. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

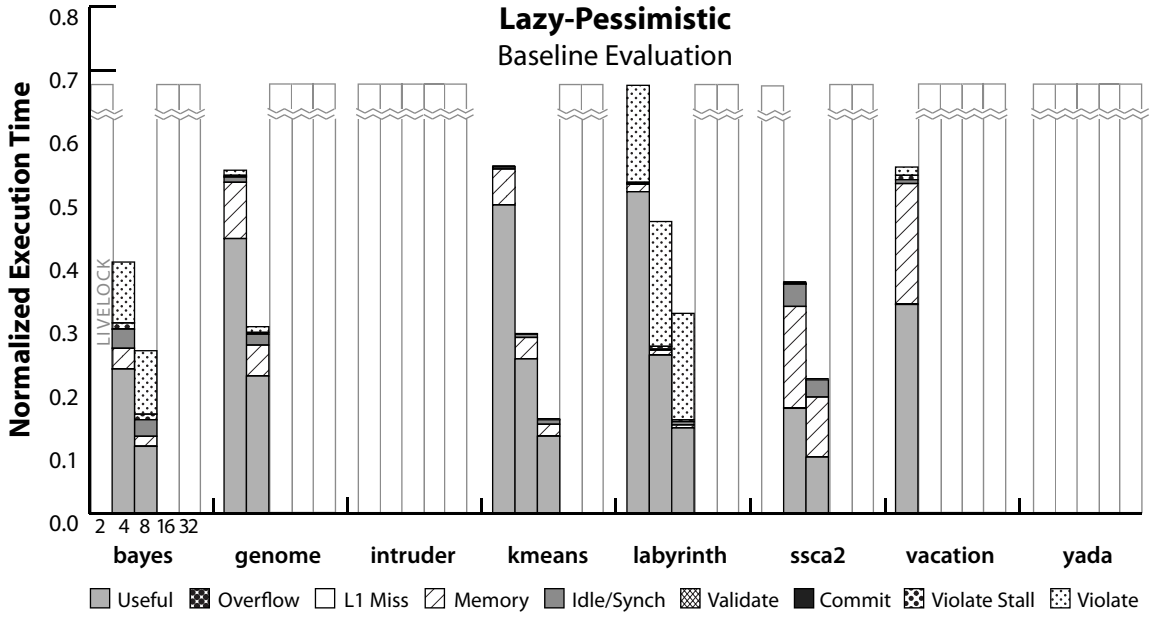

Figure 3.4: Execution time breakdown of STAMP applications on 2–32 CPUs on LP-BASE. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

Further evaluation between systems is pointless until we adjust pessimistic's contention management scheme to something more reasonable. We return to comparing the different TM systems in Section 3.6.

3.5 Contention Management in Pessimistic Conflict Detection

In Section 2.6.2, we described the pathologies and myriad solutions to address contention management issues in pessimistic conflict detection. In this section, we evaluate a handful of those solutions in an attempt to first, understand how CM policies affect the STAMP workloads on EP and LP, and second, to aid us in comparing EP and LP to LO. This should in no way be thought of as an exhaustive look at contention management.

For our limited study, we studied EP and LP with two CM policies, both with forward progress guarantees to avoid livelock: Stall and Oldest (see Section 3.2.4 for implementation details). Figure 3.5 compares speedups between the two policies on the STAMP

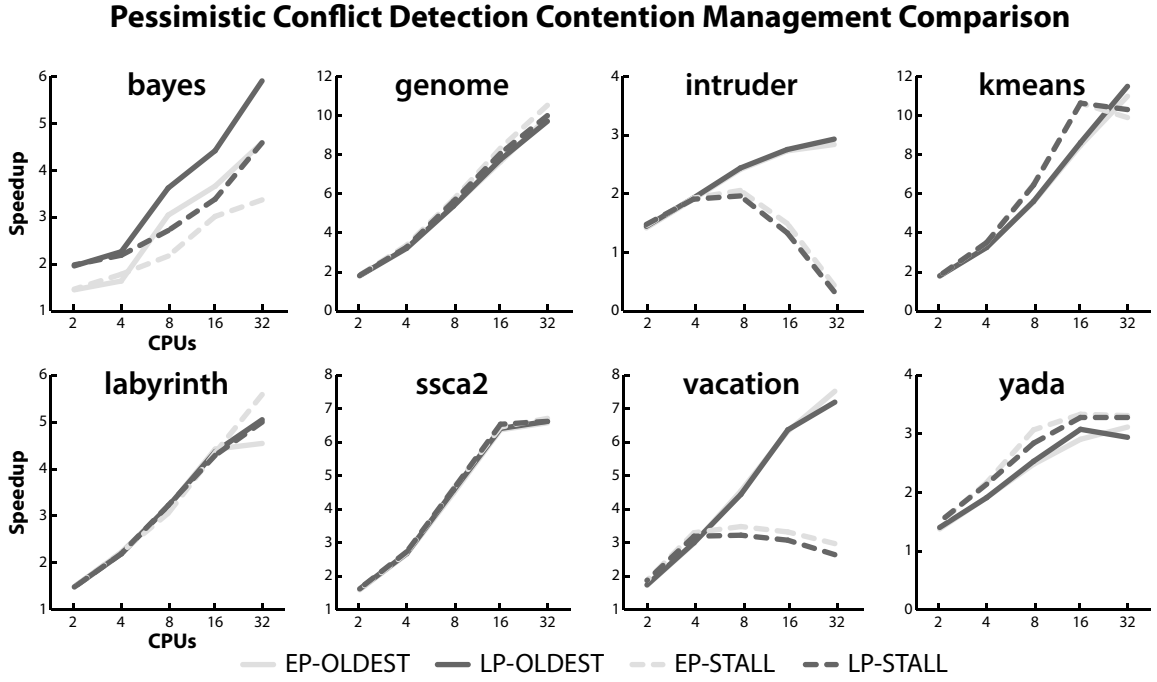

Figure 3.5: Comparing speedups of EP-OLDEST, EP-STALL, LP-OLDEST, and LP-STALL (the various contention management configurations) on the STAMP applications with 2–32 CPUs. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

applications using EP-OLDEST, EP-STALL, LP-OLDEST, and LP-STALL.

Between the policies, performance is governed by two factors: number of aborts and thread scheduling. The policy that results in fewer aborts will perform better because less work is wasted and aborts are expensive in EP. The policy whose schedule results in less load imbalance and fewer pathologies will also perform better, for obvious reasons. Unfortunately, which policy results in fewer aborts or a better transaction schedule depends on the application's transaction mix and memory access patterns and is difficult to determine *a priori*.

In our results, the STAMP applications with low-to-medium contention generally performed similarly on both the Stall and Oldest policies. On high contention benchmarks, significant differences begin to emerge between the policies.

bayes

In `bayes`, Oldest performs consistently better than `Stall` because `bayes`'s long transactions experience the FUTILESTALL pathology: transactions stalling only to later abort. In these cases, it would be better to abort and restart quickly, which a policy like Oldest encourages.

genome

`genome` has small transactions, a low contention rate, and no pathological transaction mixes. In this environment, the two policies result in very similar abort counts so they perform similarly.

intruder

`intruder` is the first application on which we see a crippling pathology in one of the policies. In this case, we see DUELINGUPGRADES in Stall: two transactions attempt to read then upgrade the same cache lines, each tripping over the other's requests. Oldest more strictly enforces priority between the feuding transactions and makes faster progress.

kmeans

`kmeans` has a medium level of contention, and we see STALL keeping abort count lower than OLDEST until 32 CPUs. At 32 CPUs, with only 15 clustering centers, it is highly probable two transactions are reading, modifying, and writing the same center. If another transaction attempts to modify that center, then a probable deadlock is detected using the conservative deadlock avoidance mechanism and an abort is triggered, increasing the number of aborts and slowing the STALL configuration.

labyrinth

`labyrinth` has a high contention rate, yet no pathologies because very few of its transactions abort. Of course, it has very large transactions (the largest by an order of magnitude compared to any other application in STAMP), so any aborts degrade performance, but there are not *interactions between aborting transactions* that would create pathologies. As with other applications where performance between STALL and OLDEST is similar, the number of aborts is similar and the performance follows.

ssca2

Like `genome`, `ssca2` has small transactions, a low contention rate, and no pathological transaction mixes. The two policies result in very similar abort counts so they perform similarly.

vacation

In `vacation`, many transactions perform reads and writes to the same data. Because of this transaction mix, Stall creates a pathology similar to STARVINGWRITER. Similar to `intruder`, Oldest enforces stricter priority, feeding the starved transactions.

yada

`yada`'s story is similar to `kmeans`: STALL reduces the number of aborts until contention becomes high, when performance becomes more similar.

3.5.1 Discussion

As was mentioned in the introduction to this section, our study is by no means exhaustive—many other CM options exist. In fact, even the policies we tried have variations and even tunable parameters (like time before retrying an access in Stall, which should probably be randomized). We did not explore these variations or parameters. It is possible that the poor performance seen with Stall in certain applications and even the pathologies in `vacation` and `intruder` could be eliminated with proper parameter tuning.

Clearly, the pessimistic conflict detection CM policy design space is large and finding the one best-performing policy across all applications may prove impossible, or at least require much experimentation. To avoid choosing one policy and dooming certain workloads to poor performance, an adaptive software approach could be taken, changing the policy based on recently seen pathologies. This is an ideal use case for our software violation handlers discussed in Section 4.2.3.

One additional policy worth attention is the suggestion of a hardware write-set predictor, as described by Bobba et al. [12]. The predictor attempts to determine what lines will eventually be upgraded with an exclusive load request. When it encounters one of

these addresses, the initial load is converted to an exclusive load, addressing the DuelingUpgrades pathology. Unfortunately, this requires extra hardware, which seems like an extreme measure just to achieve the same performance as a much simpler LO implementation.

We did not extensively evaluate this write-set predictor, but a Stanford colleague, Woongki Baek, examined a few STAMP applications with a combination of transaction age and write-set prediction and found that `intruder` is positively affected by such a policy, proving that DuelingUpgrades can be addressed.

It is also useful to note that we did not see significant LO pathologies, suggesting that LO is a more robust TM design. However, there are pathologies that can exist in LO (see Section2.6.2) and we tried a commonly advocated [12] CM policy to combat them: randomized linear backoff before aborting. Unfortunately, no applications improved and many of applications got worse. For example, `intruder` got 2.54× worse on 32 CPUs. This further proves that determining the proper CM policy and tweaking its parameters (such as maximum backoff time) is a difficult task and perhaps some adaptive method is required.

3.6 Comparing Transactional Systems

Now that we have created a relatively well-performing implementation of each of the fundamental TM designs, let's compare them to each other. Figures 3.6 and 3.7 present execution time breakdowns of the STAMP applications on the best evaluated CM configuration for each application (lower is better).

First, we notice good speedups can be achieved using any of the TM systems, with an average speedup of 7.3× on 32 CPUs across all systems and all applications (maximum of 18.9× with `kmeans` on LO). TM's non-contention related overheads were also low: little validation or commit time (except in `ssca2`) means LO's overheads were low; similar Idle/Synch times between EP and other systems means extra log writes were not creating pressure on the single-ported L1 cache. Overall, this paints a promising picture for TM's usefulness as a new parallelism paradigm. Further insights into the performance of TM relative to it's traditional MESI predecessor are gathered in Section 3.7.

Similar to our experience in the contention management experiment (Section 3.5), performance characteristics of the STAMP applications are dictated by contention level. In applications with low contention, all systems scale well and performance is similar between them. Differences between systems emerge when considering higher contention applications. Detailed results follow; speedups listed with each application are those achieved at 32 CPUs.

bayes
LO: 6.1× EP: 4.6× LP: 5.9×

While `bayes` has a high contention rate, it does scale, mainly with load imbalance (high Idle/Synch time) preventing further scalability. Load is randomly distributed across processors so convergence rates very depending on processor count and scheduling. Specifically, we see that EP's runs have more difficulty converging, suffering from vastly greater Useful time than the other two systems.

Varying convergence rates are made worse by contention: `bayes`'s transactions are very long and if one rolls back after a significant portion of its execution, that processor will be significantly behind the others in completing its work. In fact, 50% of `bayes`'s transaction conflicts on EP occurred after the transaction had executed over 13,000 cycles. Given `bayes`'s high contention rate and intrinsically random nature, this effect accumulates to acutely affect a handful of processors (in this case 2 CPUs), creating further load imbalance.

genome
LO: 12.0× EP: 10.5× LP: 10.0×

`genome`'s transactions rarely interact, given the large pool from which they pull their gene segments. This low contention rate makes results in similar performance across all three systems. However, when there are conflicts, pessimistic's early conflict detection, even tempered by the Stall policy, wastes more time in violations than optimistic.

intruder
LO: 4.5× EP: 2.8× LP: 2.9×

Because `intruder` has high contention between its many small transactions, we are not surprised that performance is limited to low CPU counts. After 8 CPUs, `intruder`

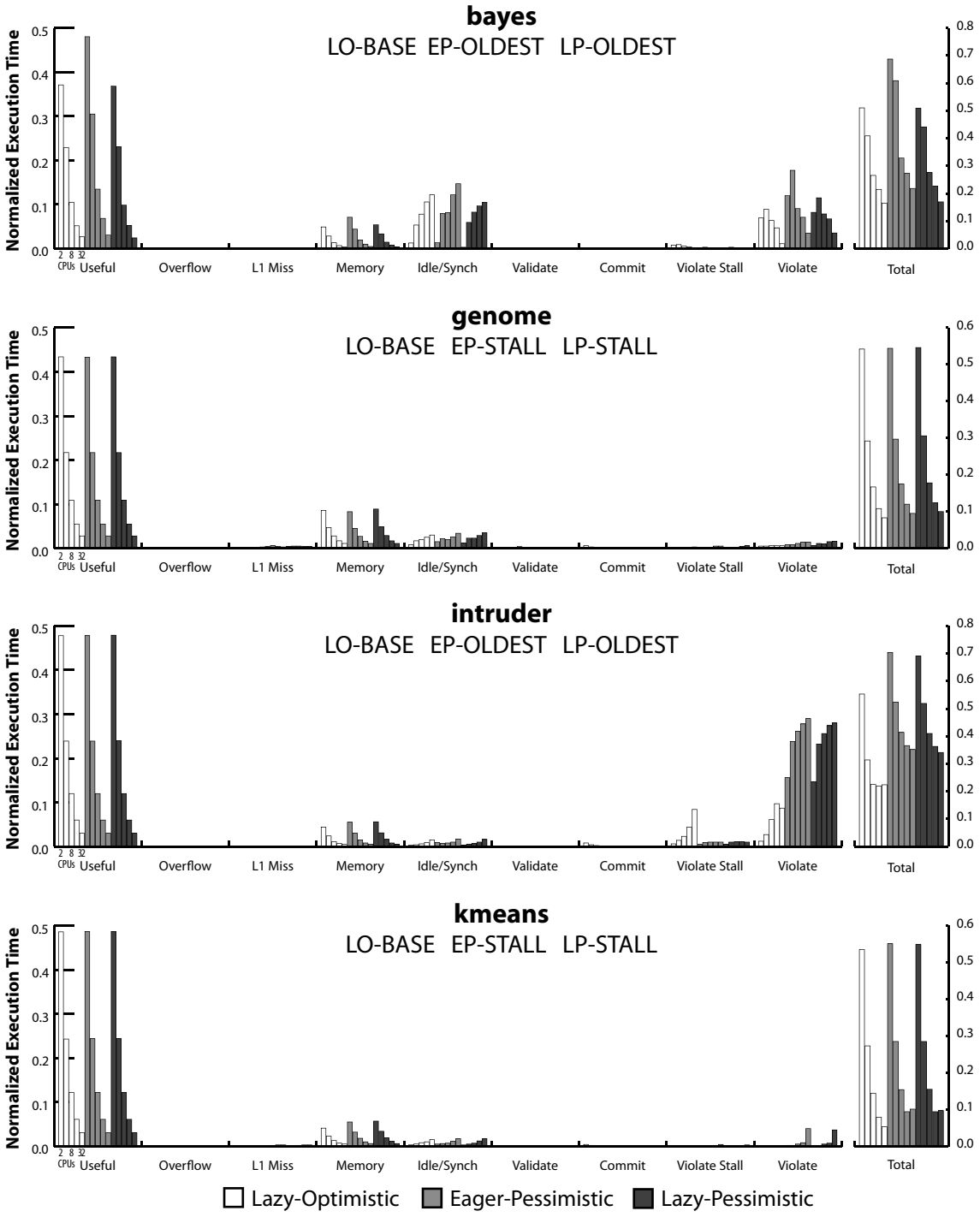

Figure 3.6: Execution time breakdown of the best evaluated contention management configuration, by application, of the first four STAMP applications on 2–32 CPUs across all systems. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

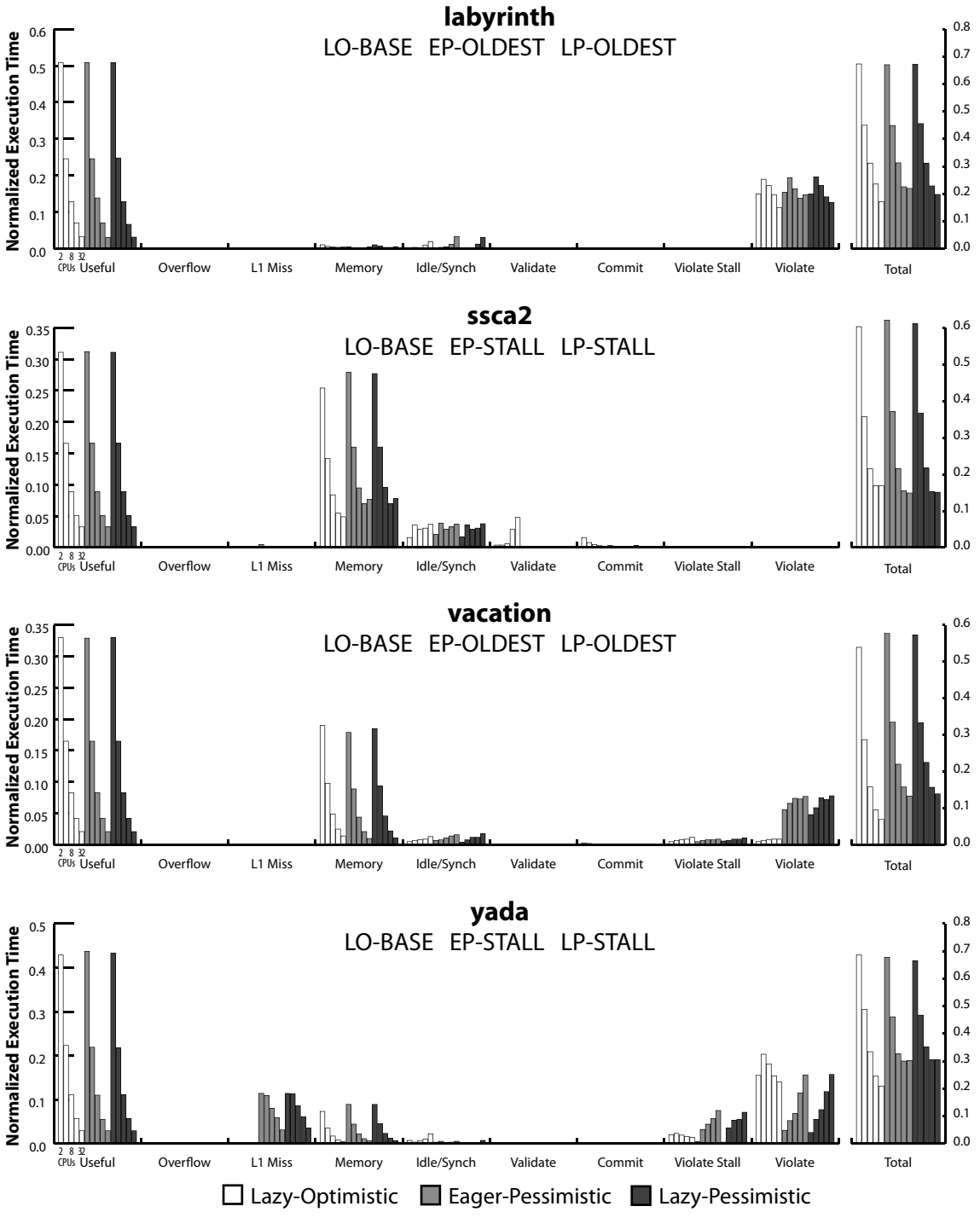

Figure 3.7: Execution time breakdown of the best evaluated contention management, by application, of the second four STAMP applications on 2–32 CPUs across all systems. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

is dominated by conflicts, which most likely come from the rebalancing of the packet assembly tree [13].

In LO, the majority of wasted time is marked as Violation Stall because these conflicting transactions are waiting on memory: most of the transactions are small (around 82 instructions) yet access somewhere between 14 and 20 cache lines worth of data. In EP and LP, `intruder` still exhibits prohibiting violations, but time is marked as Violation instead of Violation Stall because conflicts are detected early, before many of the expensive memory operations are performed. Performance still suffers because the high contention rate means there are many of these quickly aborted transactions.

As discussed in Section 3.5.1, pessimistic's performance suffers because of the DUELINGUPGRADES pathology. While Oldest does better than Stall, we would prefer to stall then use a write-set predictor to avoid DUELINGUPGRADES, since a stall-based CM policy would reduce the number of aborts.

`intruder` also exposes the impact of undoing EP's log: Violate time is significantly higher than LP's for all CPU counts. Its high writes per transactional instructions ratio combined with high contention rate, makes `intruder` particularly vulnerable to slow aborts.

kmeans

LO: 18.9× EP: 9.9× LP: 10.3×

`kmeans`'s medium contention level results in scalability on all systems but performance is much better on LO, with EP and LP both being similarly plagued by conflicts at 32 CPUs. Perhaps an additional pathology is at work in the pessimistic conflict detection systems, as evidenced by OLDEST's performance overtaking STALL's at 32 CPUs (see Section 3.5).

labyrinth

LO: 5.8× EP: 4.5× LP: 5.1×

`labyrinth`'s performance is almost identical between systems despite its high contention rate. As described in Section 3.5, aborting `labyrinth`'s large transactions contributes to its degraded performance, but no pathologies are created. We also know that `labyrinth` is sensitive to scheduling changes so we see a different system performing

better at each of 8, 16, and 32 CPUs, though the differences are small (almost imperceptible on the graphs).

ssca2
LO: 5.9× EP: 6.7× LP: 6.6×

`ssca2` is an exception to the trend that low contention applications perform similarly among the different TM systems. It has very low contention but does not scale past 16 CPUs on LO due to time spent validating. Because our implementation of LO serializes on validation, other transactions must wait to validation. This is acceptable as long as there are few transactions or the transactions are large (low write-set to instruction ratio), which will likely distribute their commit times. `ssca2` has few transactions but unfortunately, they are very small, creating high contention for commit permission. If our system supported parallel commit—allowing non-conflicting transactions to commit simultaneously—this pressure could be reduced and `ssca2` may continue to scale on LO.

EP and LP do not serialize on commit, allowing multiple non-conflicting transactions to exist simultaneously, and so continues to scale (even if slightly) until 32 CPUs. Further scaling is prevented by upgrade misses contending for the bus (visible in the Memory column), which is not experienced by LO.

vacation
LO: 14.5× EP: 7.5× LP: 7.2×

`vacation` has a similar story to `kmeans` though it has a higher contention rate. Again, Oldest seems to make poor choices about which transaction to abort, causing many wasted cycles. As discussed in Section 3.5.1, our informal experiments with write-set prediction did not seem to improve `vacation`, so by detecting conflicts too early, pessimistic conflict detection disallows a number of serializable schedules and `vacation`'s medium level of contention magnifies this effect.

yada
LO: 4.8× EP: 3.3× LP: 3.3×

`yada` performs much better on LO than on the pessimistic systems. Even though contention is high and LO suffers from a great many Violate cycles, pessimistic's Stall

policy wastes even more time. Clearly, Stall is often stalling transactions for a long time until they can proceed (evidenced by the high L1 Miss column). This would be good, saving wasted work, if all these stalled transactions eventually succeeded. However, many such transactions are subsequently violated (evidenced by many Violate cycles). For this reason, we believe `yada`-STALL may suffer from the FUTILESTALL pathology.

`yada`, like `intruder`, also exposes the effect of the undo log because its high contention rate and many writes per transaction. Unrolling the log increases Violate time on EP at 32 CPUs, further hindering scalability.

TM systems can achieve impressive speedups on parallel applications with minimal effort. When applications are parallelized using coarse-grained transactions, overheads are low, making TM a promising alternative to lock-based synchronization.

When comparing between TM systems, the largest performance differences are found on applications with significant contention. This confirms the hypothesis we made in Section 3.1.

In 7 out of 8 STAMP applications, LO performed better than the configurations of EP and LP we evaluated. In general, poor contention management (including pathologies) and additional overhead due to log unrolling limited scalability on pessimistic systems. The remaining application, despite performing better on EP and LP than on LO, does not form a strong argument in favor of pessimistic conflict detection: in `ssca2`, LO's serialized commit policy limited scalability.

3.7 Comparing to Traditional Parallelization

Unfortunately, simply improving the programming model and providing scalability is not sufficient to justify switching to HTM—after all, sequential programming is even easier and STMs also exhibit scalability. To determine whether TM is competitive with traditional synchronization, we must compare their performance.

Figure 3.8 shows the normalized execution time for our selected SPEC, SPLASH, and SPLASH-2 applications on a traditional MESI architecture using locks and done flags for

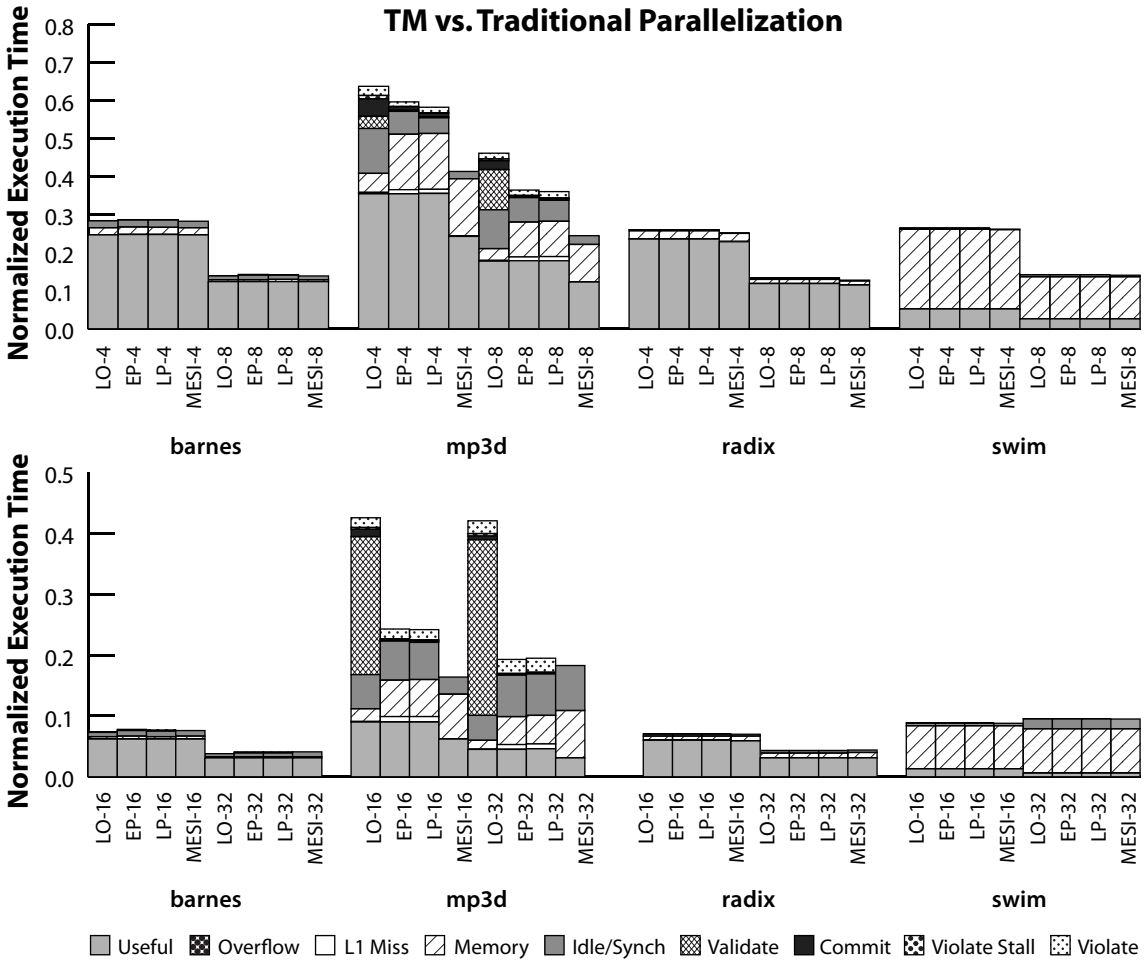

Figure 3.8: Execution time breakdown of selected SPEC, SPLASH, and SPLASH-2 applications on 4–32 CPUs using LO-BASE for all applications, EP-OLDEST and LP-OLDEST for `barnes` and EP-STALL and LP-STALL for `mp3d`, `radix`, and `swim`, and a traditional MESI parallelization. Normalized to sequential execution.

synchronization versus that of LO, EP-STALL, and LP-STALL. We chose Stall because it performed better than Oldest on all these benchmarks.

On all but one application, both TM systems and the traditional system performed similarly. We do see the expected difference in conflict detection schemes in `barnes` between LO, EP, and LP. Clearly, the early detection of pessimistic hurts performance. Not much can be said about `radix` and `swim`, both having very few transactions. It suffices to say that TM can achieve comparable speedups on these, mostly barrier-based parallel applications.

`mp3d` is a much more interesting study. First, we see that transactional overhead is more expensive than lock-overhead: the Busy portions of each TM graph are much higher than MESI's. However, we are using heavy transactions including full, unoptimized support for empty commit handlers (see Section 4.2.2). Ideally, this code would be stripped at compile time, making for much lower TM overhead.

Since it spends much of its time in transactions and has many small transactions, we see LO suffering from SERIALIZEDCOMMIT just as we did with `ssca2` in Section 3.6. Presumably, LO with parallel commit would perform similarly to the other two transactional systems on `mp3d`, just as LO does on the other benchmarks in this section.

Even though `mp3d` performs better at all CPU counts with MESI, EP and LP almost catch up at 32 CPUs. MESI begins to experience slow downs at higher CPU counts because of false sharing and global lock contention. TM will experience similar false sharing effects, but only contention that represents real shared data dependencies and not just dependencies on the same lock. As CPU counts increase, we expect performance to equalize between MESI and TM systems.

3.8 Shallow vs. Deep Memory Hierarchy

To examine how important was our choice of both a private L1 and a private L2, we evaluated the STAMP applications using only the 64-KB L1 cache. With this configuration, we found a number of applications experienced significant capacity overflows across all systems, associativity overflows being already accounted for by a victim cache (similar

to our findings in McDonald et al. [66]). Since our basic systems differ in how they handle overflow (see Section 3.2), comparing their performance has limited value. However, we present the speedups for `kmeans` and `vacation` on LO, EP-OLDEST, and LP-OLDEST compared with those configurations without private L2s in Figure 3.9.

`kmeans` represents one of the applications that experienced no capacity overflows. This is expected since it has very few transactions each with few reads and writes. We do see some degradation in performance between those runs with L2 and those without L2 due to the cost of additional capacity misses. But, since `kmeans`'s miss rates are also low, this effect is small.

`vacation` on the other hand, represents those applications that experienced significant capacity overflow—speedups are much lower in the LO and LP's runs without the private L2. This is clearly caused by the fact that LO and LP's overflow mechanisms serialize until the overflowed transaction completes. Since EP's overflow mechanism does not serialize, its L1 performance is almost identical to that of its L1+L2 performance. Note that LP-L1 performs so much better at 32 CPUs than at 16 CPUs because the extra contention causes conflicts, and therefore aborts, to happen sooner, reducing the number of overflows and, in turn, the amount of serialization.

It is instructive to look closer at `vacation`, which exhibits the same problems seen, at various severities, in other applications. Figure 3.10 presents the execution time breakdown of `vacation` on LO without private L2 for 2–32 CPUs. Clearly, overflow cycles play a significant role in diminishing performance, slowly relinquishing their hold as overall cache capacity increases with larger CPU counts. Even though fewer transactions overflow, performance remains much the same between 16 and 32 CPUs due to increasing Validation time caused by LO's serializing overflow implementation.

These results should not be interpreted as a direct comparison between LO, EP, and LP, but rather to support the need for efficient virtualization mechanisms. As we discussed in Section 2.7, overflow implementations are affected by choice of TM system, but are not inherent to them and more efficient implementations for our lazy versioning systems have been proposed.

In conclusion, for TM to perform competitively, adequate speculative storage must

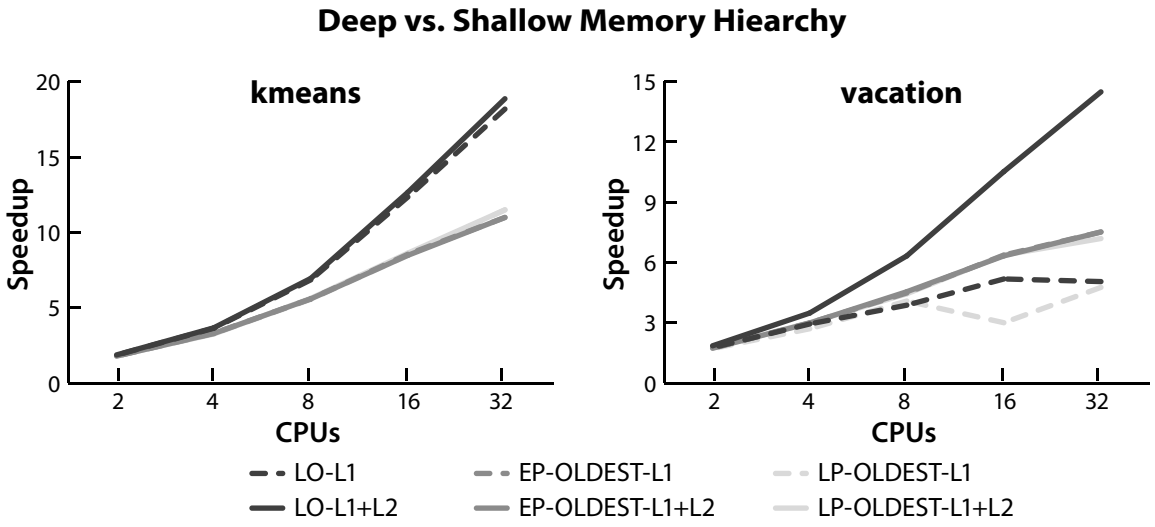

Figure 3.9: Speedups of `kmeans` and `vacation` on 2–32 CPUs on LO, EP-OLDEST, and LP-OLDEST with and without a private L2. L1 runs normalized to sequential execution with only L1; L1+L2 runs normalized to sequential execution with both an L1 and an L2. Note that on both graphs, the EP lines are almost collinear, and on `kmeans`, the LP lines are almost collinear.

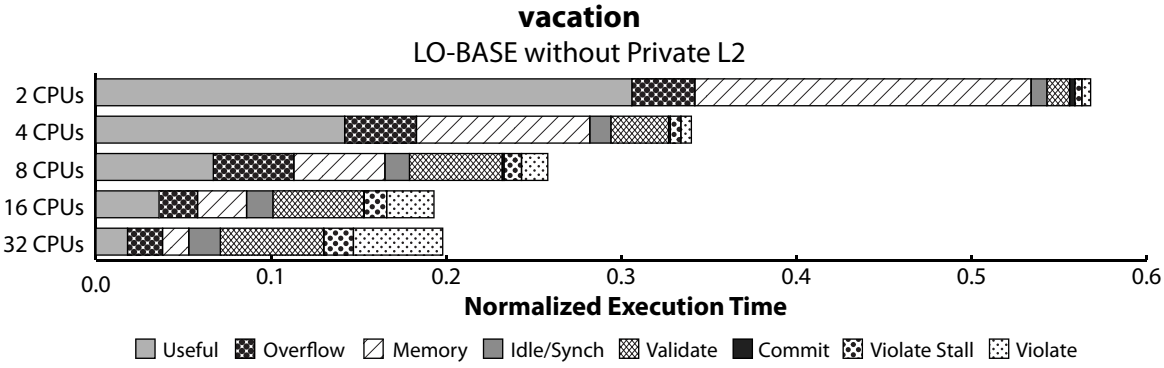

Figure 3.10: Execution time breakdown of `vacation` on 2–32 CPUs on LO, but without a private L2. Normalized to sequential execution with only L1.

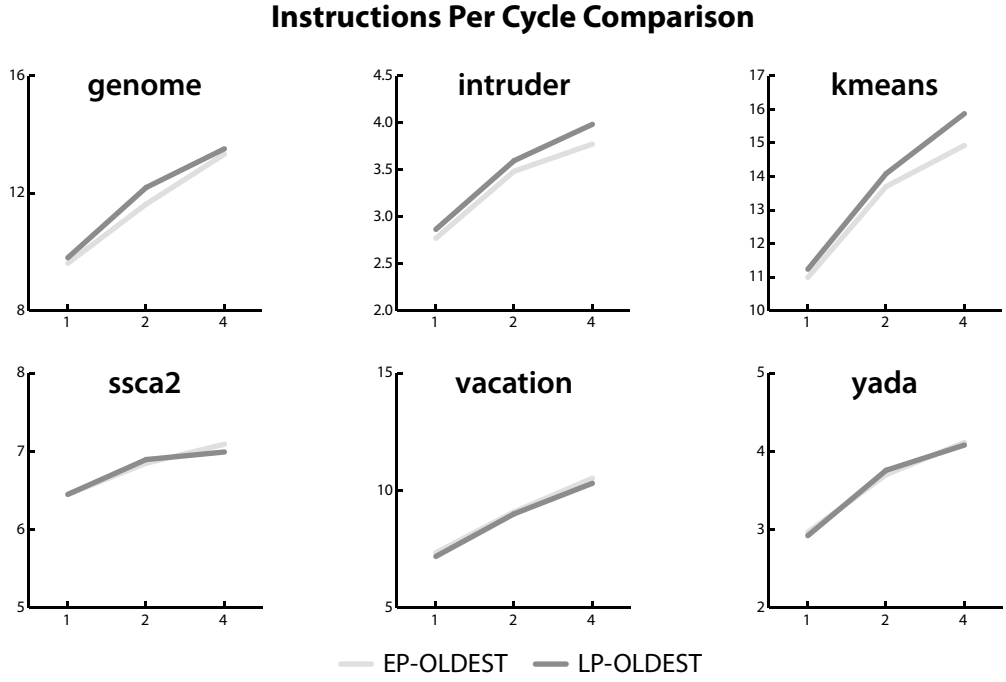

Figure 3.11: Speedups of all three systems on a single issue, a 2-issue, and a 4-issue CPU. All applications are presented at 32 CPUs. The Oldest CM policy is used for pessimistic systems. Speedups relative to sequential, single issue execution.

be available or a reasonable overflow scheme must be employed. Not surprisingly, victim caches, while shown to be useful in avoiding associativity overflows [66], are not sufficient for avoiding capacity overflows.

3.9 Instruction-Level Parallelism

We speculated that EP's additional log writes did not degrade performance in our earlier experiments because our simulated machine issues only one instruction per cycle. In other words, because back-to-back speculative writes are rare, computation instructions hid the fact that the single-ported L1 was also serving log writes.

To test the sensitivity of TM designs, especially EP, to a more realistic CPU, we simulated a multi-issue machine by continuing to execute instructions in a single cycle until either an IPC limit was reached, or an instruction accessed memory. We simulated IPCs

of 1, 2, and 4, but attempted no instruction re-ordering. Figure 3.11 shows speedups on all three systems with these configurations. We chose OLDEST as our pessimistic conflict manager because it produced the most consistently acceptable results in Section 3.5.

When examining Figure 3.11, do not consider the absolute speedup values at each configuration. Instead, compare the *shape* of the curves: if two systems exhibit a similar shaped curve, then we can conclude that one system is not more affected by IPC than the other. However, if the change in speedup is less for one system than for another, the first system is experiencing bottlenecks exposed by high IPCs. For this experiment, we focused on the differences between the EP curve and LP curve, which should highlight the impact of log writes while factoring out contention management effects.

Examining the results, we found that most applications with high ratios of writes to instructions (listed as "writes per 1K instructions per transaction" in Table 3.2) did experience slightly slowed speedup at higher IPCs on EP versus LP. In these applications, we find many cycles where the processor issues a store, but must stall because the L1 is busy processing a log write. `bayes` and `labyrinth` were excluded from this experiment because varying the IPC drastically changed the transaction scheduling and convergence rates, making conclusions about how IPC affects overheads impossible. Detailed results follow.

genome

`genome` has a small writes per 1,000 instructions ratio (7.1), so we expect EP's log writes to have little impact on performance. Indeed this is the case, with EP performing only slightly worse than LP due to its more expensive aborts, just as we see in Section 3.6. Because EP's log unroller also benefits from higher IPC, we see the gap between EP and LP's performance close as IPC reaches 4.

intruder

`intruder` has a high writes per 1,000 instructions ratio (31.6), so we begin to see a difference between EP's speedup and LP's speedup at higher IPCs. .5% of cycles in EP-IPC4 were spent waiting for an unavailable L1 cache compared to only .3% of cycles in LP-IPC4. While this demonstrates the sensitivity of EP to higher IPCs, LP outperforms

EP by only 5%, so perhaps the extra log writes are not of great concern.

kmeans

`kmeans` has a similar story to `intruder`: a high writes per transactional instruction ratio makes speedup suffer on EP at higher IPCs. Again though, the performance difference between EP and LP is only about 6%.

Because higher IPC makes transactions shorter, contention increases for commit permission in LO, finally reaching a point of degrading performance at IPC=4. This is the same SerializedCommit pathology we saw in `ssca2` in Section 3.6, further justifying the need for parallel commit in LO systems.

ssca2

`ssca2` is an exception: it has the highest writes per 1,000 instructions ratio, yet EP performs better than LP at IPC=4, if only 1.4% better. So, despite spending 1.6% more of its cycles waiting for the L1, EP outperforms LP because `ssca2` has few transactions and waiting while unrolling the log acts as a backoff mechanism to improve load imbalance (see STALL's slight lead over OLDEST in Figure 3.5).

vacation

Performance was almost identical between EP and LP on `vacation`. It has only 7.7 writes per 1,000 instructions, which is quite low, so we expected increased IPC to have little impact.

yada

`yada` has an even lower writes per 1,000 instructions ratio than `vacation`, so its performance is also roughly identical between EP and LP.

In conclusion, while EP does show some sensitivity at high IPC values, only two applications (`intruder` and `kmeans`) demonstrated performance degradation on EP. Even that difference was small (about 5%) and only obtained at IPC=4, a rather high number considering most recent multi-core machines are dual-issue. In conclusion, the additional pressure EP's log writes exert on a single-ported L1 cache are not of great concern.

3.10 Interconnect Parameters

Similar to our concerns about higher IPC and EP's log writes, we speculated that LO may be especially sensitive to bus bandwidth and latency because it broadcasts its write-set addresses on commit. We also had concerns about pessimistic's use of the bus to acquire immediate write permission on speculatively modified cache lines. To test these hypotheses, we evaluated the performance of all three systems with 32 CPUs, first increasing arbitration time, then in a separate experiment, reducing the available bandwidth. For these experiments, we chose the OLDEST contention manager for EP and LP since it produced the most consistently acceptable results across all applications.

First, we increased the arbitration time from 3 cycles to 6 cycles. Performance degradation averaged 10.8% and varied from .29% (EP `yada`) to 31.1% (EP `ssca2`). Strangely, performance improved as much as 17.2% (EP `bayes`) on select configurations in certain applications because fortuitous scheduling reduced load imbalance (LO: `labyrinth`, EP: `bayes` and `labyrinth`).

The cause of performance differences varied between applications and between systems. For further investigation, we examine the execution time breakdown for three representative cases: `intruder`, `labyrinth`, and `ssca2`. Figure 3.12 presents the results.

`intruder`, with its long transactions full of memory accesses, experienced greater Violate Stall time in LO because that system waits to detect conflicts until commit time, after many accesses made by other transactions. EP and LP experienced similar performance degradation but the cause was having to wait longer to detect conflicts, causing an increase in the Violate time (see `intruder`'s performance with the Stall policy in Section 3.5). `vacation`'s performance pattern was similar.

On `ssca2`, an application with a high global L2 miss rate, all systems experienced additional time spent sending refill requests across the bus (represented by increased Memory time). Increased Validate and slightly increased Commit time show the sensitivity of LO to increased arbitration time. This is a validation of our hypothesis that LO's overheads are affected by interconnect characteristics. However, as we established in Section 3.6, since `ssca2` suffers from the SerializedCommit pathology, this effect would be mitigated by parallel commit. `labyrinth` represents two classes of applications. First,

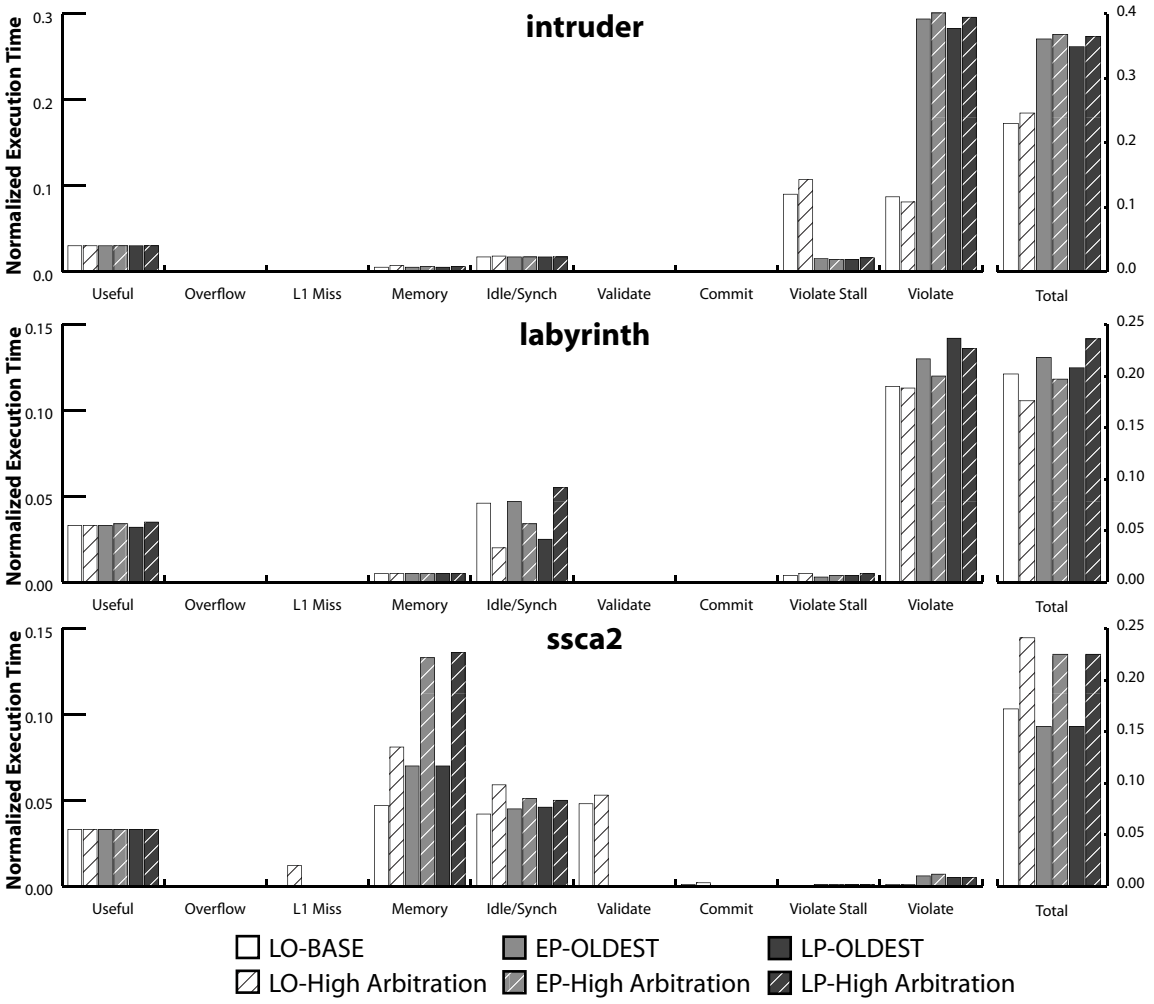

Figure 3.12: Execution time breakdown, on all three systems, of selected STAMP applications using both 3-cycle and 6-cycle arbitration time on 32 CPUs. Normalized to sequential execution with 3-cycle arbitration time.

we see increased load imbalance (Idle/Synch time) in some configurations due to scheduling changes (LP in this case). Secondly, `labyrinth` and `bayes` exhibit the strange effect of certain systems improving their performance. This happens when significant load imbalance exists initially and a change in transaction scheduling reduces this imbalance. Unfortunately, both `genome`, `kmeans`, and `yada` exhibited increased load imbalance like these applications, but not the fortuitous scheduling.

Next, we reduced the bus bandwidth from 32B per cycle (a full cache line per cycle) to 16B per cycle. Performance degraded an average of 6.5% and varied from almost 0% to 42.4%. Again, we saw some configurations improve their performance. As expected, performance did not degrade or improve as much as in the arbitration experiment, since many packets do not require much bandwidth (such as load requests) but all requests must arbitrate for the bus.

The results are quite similar to the arbitration experiment, only with decreased effects, so we do not present any detailed breakdowns. In fact, `intruder` and `vacation` experienced only slightly more Violate Stall on LO and Violate on EP and LP. `ssca2` did experience significant additional memory stall since refills now require an additional cycle to cross the bus. LO overhead was slightly higher, but even less significant than in the arbitration experiment. Because `ssca2` writes few lines per transaction, bandwidth is not the limiting factor, instead it is the number of commit packets sent, which is more affected by arbitration time. Again, some applications (`labyrinth` and `yada`) improved their performance due to scheduling differences, while other applications (`bayes`, `genome`, and `kmeans`) experienced increased load imbalance.

In conclusion, we observed only a slight sensitivity in LO-specific overhead to higher arbitration time or lower bandwidth, and that only when Commit or Validation time is already significant. Reasonable arbitration and bandwidth limits are more than sufficient to achieve reasonable performance from all TM systems, including LO. Finally but not surprisingly, interconnect latency has a larger impact on overall performance than bandwidth in TM systems.

3.11 Related Work

As mentioned in Section 3.5, Bobba et al. [12] also examined the three fundamental TM design points. They evaluate the systems using some SPLASH applications (including `barnes` and `mp3d`) and two microbenchmarks, and come to many similar conclusions: differences arise between systems mainly in high contention applications and dealing with CM is the most important element in determining performance.

However, they try significantly more complex CM schemes to address the pathologies seen among pessimistic conflict detection systems for their LogTM design. Specifically, they find that using a hardware write-set predictor alone or combining it with a timestamp scheme (similar to OLDEST) performs best for EP. From their results, it seems that LO performs almost as well or better than other options except on `mp3d` where the complex EP schemes requiring the additional hardware write-set perform significantly better. This is similar to our results: LO performs very well despite its simplicity.

Ceze et al. [19] compare their transactional Bulk architecture to an LO and an EP system across applications from the Java Grande Forum [55] benchmark as well as SPECjbb-2000 [98]. In general, they find the two systems perform quite similarly except on SPECjbb2000. In that application, LO performs significantly better than EP due to the DuelingUpgrades pathology and the fact that pessimistic detects conflicts that eventually result in a serializable schedule.

Cao Minh et al. [13], while introducing the STAMP benchmarks, also compares their performance on both an LO and EP HTM as well as various STMs. They find, as we do, that EP experiences significant contention management problems and that LO generally performs the better. They also show that simple virtualization techniques, like serializing on overflow, are not adequate.

FlexTM [92] employs a set of architectural components to build a TM system of your choice, including choice between conflict detection schemes. FlexTM builds on their earlier work [93] which decouples version management and conflict detection but requires software metadata handling. From their experiments, using microbenchmarks and `vacation`, they conclude that simple performance is superior with optimistic. They

perform an additional experiment however, designed to evaluate the two conflict detection schemes on a timesharing system where thread switching occurs on transactional abort. Pessimistic's early conflict detection proves to outperform optimistic in throughput tests with microbenchmarks because optimistic waits to detect conflicts. We did not evaluate virtualization of TM systems; see Section 2.7 for a discussion of how to virtualize the different TM systems, including better options for virtualizing LO.

Numerous studies have been published comparing various TM designs to traditional parallelization techniques. Their conclusions are similar to ours, namely that performance is similar or better between TM and traditional, lock-based synchronization, making TM an attractive alternative.

Ananian et al. [8] proposed two TM designs: Unbounded Transactional Memory (UTM, an EP design) and Large Transactional Memory (LTM, an LP design), with a focus on solving challenges with transactional virtualization. They compare LTM to a traditional lock-based architecture using a shared counter microbenchmark and transactionalized versions of the Linux kernel and some SPECjvm98 benchmarks. They convert locks and `synchronized` regions in transactions, similar to our technique. On the shared counter microbenchmark, they found that LTM scales better, despite having very small transactions, because their load-linked, store-conditional locks introduce much unneeded network traffic and cache invalidations.

On their larger benchmarks, they evaluate lock overhead versus TM overhead, finding TM introduces far less execution time (typically under 10%) than spinlocks (sometimes as much as 75%). They attribute this pleasing result to the fact that, in their transactional conversion, nested locks are subsumed into one transaction. They assume their UTM design will perform similarly to their LTM design, though they do not evaluate it.

Moore et al. [69] compared LogTM to a traditional system using microbenchmarks and SPLASH-2. First, they examine LogTM versus various locking schemes on a shared counter benchmark, finding that TM continues to scale far beyond the lock-based implementations, further supporting claims made by the authors cited above. For their SPLASH-2 applications, they also replace locks with transactions, just as we did. They

find that LogTM performs better than their traditional parallelization on every application, citing TM's optimistic concurrency. Our results are not as stellar because of our unoptimized `tm_begin()` and `tm_end()` functions.

Recently, researchers have proposed two extensions to conventional parallel machines that target some of its performance bottlenecks. Coherence Decoupling (CD) [53] allows a processor to use invalid data in the cache while coherence messages are exchanged over the interconnect. This reduces the performance impact of false sharing and silent stores. The CD evaluation shows a 1% to 12% performance improvement for SPLASH applications on a 16 processor SMP. Speculative Synchronization (SS) [79, 64] allows a processor to speculatively proceed past locks and barriers and achieve the benefits of optimistic non-blocking synchronization. It provides a 5% to 25% performance improvement for SPLASH applications on a 16 processor SMP [79].

For the CMP environment in this study, false sharing, silent stores, and synchronization pose smaller performance challenges than with a SMP system, as the on-chip interconnect in a CMP has higher bandwidth and lower latency. Hence, the benefits from CD and SS in a CMP will be significantly lower and are unlikely to change the comparison between TM and MESI.

For our evaluation, MESI used sequential consistency and all loads and stores from each processor in both systems are strictly ordered. The use of a relaxed consistency model would undoubtedly improve performance [2]. Nevertheless, TM would also improve as we can freely reorder loads and stores within transactions. In fact, TM with an out-of-order processor can be thought of as implementing release consistency, with transaction begin and end being acquire and release, respectively.

Only a handful of researchers have examined the sensitivity of TM designs to interconnect parameters. Hammond et al. [39] performed preliminary, trace-driven evaluations of an ideal TCC system (which is LO with continuous transactions; see Chapter 5) using a number of benchmarks from various suites include SPLASH-2, SPECjvm98, SPEC CFP, and SPECjbb2000. They expected that LO's bandwidth requirements, while

higher than what has been required for traditional CMPs, were in fact reasonable.

As a follow up to that study, we performed a more rigorous, execution-driven evaluation in McDonald et al. [66]. Through execution-driven experiments, we confirmed the conclusion of Hammond et al., that even with the added burden of continuous transactions, LO's bandwidth and latency sensitivity is not a concern.

We did not directly compare bandwidth usage between TM systems, however Ceze et al. [19] found that, for their TM implementations, EP used more bandwidth than LO on half of their benchmarks (because of additional invalidations and subsequent refills), and LO used more bandwidth on the remainder of applications (because of commit requirements). Despite the fact that The average across all applications showed LO using slightly less bandwidth, overall bandwidth requirements for both systems were reasonable in a modern CMP environment.

3.12 Conclusions

The biggest performance differences between systems are to be found on applications with high contention and therefore, contention management becomes critical to performance. Furthermore, pessimistic conflict detection presents the most complicated CM needs in the applications we examined. Even though we found well performing versions of pessimistic conflict systems by varying CM policies, it is difficult to choose a one best CM policy for all applications. Finally, even though we did not fully explore pessimistic's CM design space, the fact that the basic CM policy for LO performed as good or better than EP and LP in most of our benchmarks suggests that LO (with the addition of parallel commit) is the preferred TM system from a performance and implementation complexity standpoint. We believe the work of others also supports this conclusion (see Section 3.11).

Speculative state storage is critical to performance and in the cases where storage is insufficient, overflow management is important. Though we did not evaluate different overflow mechanisms, a non-serializing option is clearly best since performance suffers greatly without it.

EP's log writes are not a great concern because they are generally cached and, while

they do introduce additional pressure on cache ports at high IPC, this did not result in a significant performance bottleneck.

Likewise, LO's additional latency and bandwidth requirements are not a great concern either. We saw only one application where any overhead-specific effect was seen and that application has an overhead pathology with a known solution (`ssca2`'s serialized commit).

Finally, comparing TM to traditional parallel systems reveals that TM is a viable alternative to lock-based parallelization. Most applications we evaluated performed similarly between a MESI-based system and our three TM systems. However, TM designers must be careful to minimize the overhead associated with creating and destroying a transaction if similar performance is to be expected from a parallelization where locks have simply been replaced with transactions.

Chapter 4

The Architectural Semantics of HTM

All the uses of TM and the exact semantics required to implement languages and operating systems are not clear and will not be until TM is firmly established. However, for research to take place, systems must have flexible and powerful tools with which to experiment. STMs provide the most flexible environment, able to implement any semantics on any hardware system, but this flexibility comes at a hefty performance cost. Instead, we would like to define minimal hardware constructs (and software conventions) easily implemented in an HTM.

At the instruction set level, the HTM systems presented above provide only a few instructions to define transaction boundaries. While such limited semantics have been sufficient to demonstrate HTM's performance potential using simple benchmarks, they fall short of supporting several key aspects of modern programming languages and operating systems such as transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions. Moreover, simple HTM semantics are insufficient to support recently proposed languages and runtime systems that build upon transactions to provide an easy-to-use concurrent programming model [40, 42, 22, 6, 30, 83, 62, 35, 1, 17]. For HTM systems to become useful to programmers and achieve widespread acceptance, it is critical to carefully design expressive and clean interfaces between transactional hardware and software.

This chapter defines a *comprehensive instruction set architecture (ISA) for hardware*

transactional memory. The architecture introduces four basic mechanisms: (1) *two-phase transaction commit*, (2) *support for software handlers on transaction commit, violation, and abort*, (3) *closed- and open-nested transactions with independent rollback*, and (4) *non-transactional loads and stores*. Two-phase commit enables user-initiated code to run after a transaction is validated but before it commits in order to finalize tasks or coordinate with other modules. Software handlers allow runtime systems to assume control of transactional events to control scheduling and insert compensating actions. Closed nesting is used to create composable programs for which a conflict in an inner module does not restrict the concurrency of an outer module. Open nesting allows the execution of system code with independent atomicity and isolation from the user code that triggered it. Non-transactional loads and stores allow transactions to avoid generating spurious dependencies on data known to be private. The proposed mechanisms require a small set of ISA resources, registers and instructions, as a significant portion of their functionality is implemented through software conventions. This is analogous to function call and interrupt handling support in modern architectures, which is limited to a few special instructions (e.g., jump and link or return from interrupt), but rely heavily on well-defined software conventions.

We demonstrate that the four proposed mechanisms are sufficient to support rich functionality in programming languages and operating systems including transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions within transactions. We also argue that their semantics provide a solid substrate to support future developments in TM software research. We describe practical implementations of the mechanisms that are compatible with HTM architectures. Specifically, we present the modifications necessary to properly track transactional state and detect conflicts for multiple nested transactions. Using execution-driven simulation, we evaluate I/O and conditional synchronization within transactions.

Overall, this chapter is an effort to revisit concurrency support in modern instruction sets by carefully balancing software flexibility and hardware efficiency. Our specific contributions are:

- We propose the first comprehensive instruction set architecture for hardware transactional memory that introduces support for two-phase transaction commit; software handlers for commit, violation, and abort; closed- and open-nested transactions with independent rollback; and non-transactional loads and stores.
- We demonstrate that the three proposed mechanisms provide sufficient support to implement functionality such as transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions within transactions. No further concurrency control mechanisms are necessary for user or system code.
- We implement and quantitatively evaluate the proposed ISA. We show that overheads for these mechanisms are reasonable. We also demonstrate scalable performance for transactional I/O and conditional scheduling.

4.1 The Need for Rich HTM Semantics

Basic HTM systems provide instructions to define transaction boundaries, which is sufficient to support programming constructs such as `atomic{}` and demonstrate the HTM performance potential with simple benchmarks. However, these systems fall short of supporting key aspects of modern programming environments [60]. Moreover, there is a significant body of work on languages and runtime systems that builds on transactions to provide an easy-to-use concurrent programming model [40, 42, 22, 6, 30, 83, 62, 35, 1, 17, 16]. To gain the flexibility of STM semantics but with superior performance, HTM systems must support a full programming environment and allow for innovation in transaction-based software. This section reviews the basic software requirements that motivate the semantics proposed in Section 4.2.

Composable Software (libraries): Modern programs use hierarchies of libraries, which have well-defined interfaces, but their implementation is hidden from users. Since libraries called within transactions may include atomic blocks, transactions will often be nested. One simple way to deal with nested transactions is by subsuming (or flattening) all inner transactions within the outermost one [39, 8, 69]. Flattening can hurt performance significantly as a conflict in a small, inner transaction may cause the re-execution

of a large, outer transaction. Such conflicts may even occur due to bookkeeping data maintained by the library, which are only tangentially related to the concurrency in the overall program. To avoid such bottlenecks without nesting, a programmer must be aware of the implementation details of the library code, which is completely impractical. Hence, HTM systems must support independent abort of nested transactions.

Contention and Error Management: As seen previously, it is beneficial to extend simple abort and re-execute semantics with contention managers. We simulated their execution in software which is well suited to improve performance and eliminate starvation [35]. But, there must be an infrastructure for executing this software when conflicts are detected by hardware. Language constructs such as `tryatomic` [6] and `ContextListener` [40] allow alternate execution paths on transaction aborts. With nested transactions, programmers may define separate conflict policies for each nesting level. Finally, it is necessary for both error handling (e.g., `try/catch`) and debugging to expose some information about the aborted transaction before its state is rolled back. Hence, HTM systems must intervene on all exceptional events and manage transactional state and program control flow.

Conditional Synchronization: Transactions must replace locks not only for atomic execution but also for conditional synchronization (e.g., `wait/notify`). Conditional synchronization is useful with producer/consumer constructs and efficient barriers. Recently proposed languages include a variety of constructs that build upon transactions to provide conditional synchronization without explicit `notify` statements (conditional `atomic` [41], `retry` and `orElse` [42], `yield` [83], `when` [22], `watch` and `retry` [17]). To support such constructs, software needs control over conflicts and commits, and HTMs must also facilitate communication between uncommitted transactions.

System Calls, I/O, and Runtime Exceptions: Simple HTM systems prohibit system calls, I/O, and runtime exceptions within transactions [8, 69] or revert to sequential execution on such events [39]. Both approaches are unacceptable as real programs include system calls and cause exceptions, often hidden within libraries. To avoid long transactions through system code invoked explicitly (system calls) or implicitly (exceptions), system code should update shared-memory independently of the user transaction that

triggered it. There should also be mechanisms to postpone system calls until the corresponding user code commits or compensate for the system call if the corresponding user code aborts. Furthermore, system programmers should be able to use the atomicity and isolation available with transactional memory to simplify system code development.

4.2 HTM Instruction Set Architecture

To provide robust support for user and system software, we introduce four key mechanisms to HTM architectures: *two-phase commit*, *transactional handlers*, *closed- and open-nested transactions*, and *non-transactional loads and stores*. This section describes their ISA-level semantics and introduces the necessary state and instructions. We discuss their implementation in Section 4.4. The instructions should be used by language and system developers to implement high-level functionality that programmers access through language constructs and APIs. The instructions provide key primitives for flexible software development and do not dictate any end-to-end solutions.

Throughout this section, we refer to Tables 4.1 and 4.2 that summarize the state and instructions necessary for the four mechanisms. Some basic instructions are already available in some form in HTM systems (e.g., `xbegin`, `xabort`, `xregrestore`, and `xrwsetclear`), but we need to modify their semantics. The exact encoding or name for instructions and registers depends on the base ISA used and is unimportant.

Each transaction is associated with a *Transaction Control Block (TCB)* in the same manner as a function call is associated with an activation record. The TCB is a logical structure that stores basic transaction state: a status word, the register checkpoint at the beginning of the transaction (including the PC), the read-set and write-set addresses, and the writebuffer or undo log. Conceptually, all TCB fields can be stored in cacheable, thread-private main memory. In practice, several TCB fields will be in caches (e.g., the read-set, write-set, and writebuffer/undo log) or in registers (e.g., the status word) for faster access. Figure 4.2 summarizes the final view of the TCB.

	BASIC STATE	
`xstatus`	reg	Transaction info: ID, type (closed, open), status (active, validated, committed, aborted, or waiting to abort), nesting level
`xtcbptr_base`	reg	Base address of TCB stack
`xtcbptr_top`	reg	Address of current TCB frame
	HANDLER STATE	
`xchcode`	reg	PC for commit handler code
`xvhcode`	reg	PC for violation handler code
`xahcode`	reg	PC for abort handler code
`xchptr_base`	TCB	Base of commit handler stack
`xchptr_top`	TCB	Top of commit handler stack
`xvhptr_base`	TCB	Base of violation handler stack
`xvhptr_top`	TCB	Top of violation handler stack
`xahptr_base`	TCB	Base of abort handler stack
`xahptr_top`	TCB	Top of abort handler stack
	VIOLATION & ABORT STATE	
`xvPC`	reg	Saved PC on violation or abort
`xvaddrs[`$0...m$`]`	TCB	List of conflicting addresses
`xvcount`	TCB	Size of `xvaddrs`
`xvoverflow`	TCB	Bit set if `xvaddrs` overflows
`xvaddrs_s[`$0...n$`]`	reg	Hardware's shadow version of `xvaddrs`
`xvlevels_s[`$0...n$`]`	reg	`xvaddrs_s[`i`]` is a conflicting address for nesting level `xvlevels_s[`i`]`
`xvcount_s`	reg	The current length of `xvaddrs_s` and also `xvlevels_s`.
`xvoverflow_s`	reg	Bit set if `xvaddrs_s` overflows

Table 4.1: State needed for rich HTM semantics. State may be in a processor register or stored in a TCB field.

	TRANSACTION DEFINITION
`xbegin`	Checkpoint registers & start (closed-nested) transaction
`xbegin_open`	Checkpoint registers & start open-nested transaction
`xvalidate`	Validate read-set for current transaction
`xcommit`	Atomically commit current transaction
	STATE & HANDLER MANAGEMENT
`xrwsetclear`	Discard current read-set and write-set; clear `xvpending` at current nesting level
`xregrestore`	Restore current register checkpoint
`xabort`	Abort current transaction; jump to `xahcode`; disable further violation reporting
`xvret`	Return from abort or violation handler; jump to `xvPC`; enable violation reporting
`xenviolrep`	Enable violation reporting
	NON-TRANSACTIONAL OPERATIONS
`imld`	Load without adding to read-set
`imst`	Store to memory without adding to write-set
`imstid`	Store to memory without adding to write-set; no undo information maintained

Table 4.2: Instructions needed for rich HTM semantics.

4.2.1 Two-phase Commit

Semantics: We replace the monolithic commit instruction in with *two-phase commit* [34]. The `xvalidate` instruction verifies that atomicity was maintained (i.e., no conflicts) and sets the transaction status to *validated*. Its completion specifies that the transaction will not be rolled back due to a prior memory access. The `xcommit` instruction marks the transaction *committed*, which makes its writes visible to shared memory. Any code between the two instructions executes as part of the current transaction and has access to its speculative state. The code can access thread-private state safely, but accesses to shared data may cause conflicts and should be wrapped within open-nested transactions (see Section 4.2.5). The code can also initiate voluntary aborts instead of an `xcommit`.

Use: Two-phase commit allows the compiler or runtime to insert code between `xvalidate` and `xcommit`. This is useful for commit handlers that help finalize system calls and I/O. It also enables the transaction to coordinate with other code before it commits. For example, we can run a transaction in parallel with code that checks its correctness (e.g., for memory leaks, stack overflows, etc.) [77]. Alternatively, we can coordinate multiple transactions collaborating on the same task for group commit [62].

4.2.2 Commit Handlers

Semantics: Commit handlers allow software to register functions that run once a transaction is known to complete successfully. Commit handlers execute between `xvalidate` and `xcommit` and require no further hardware support. Everything else is flexible software conventions. It is desirable that transactions can register multiple handlers, each with an arbitrary number of arguments. Hence, we define a *commit handler stack* in thread-private memory. The base and top of the stack are tracked in the TCB fields `xchptr_base` and `xchptr_top`, respectively. To register a commit handler, the transaction pushes a pointer to the handler code and its arguments on the stack. An additional TCB field, (`xchcode`), points to the code that walks the stack and executes all handlers after `xvalidate`. To preserve the sequential semantics of the transaction code, commit handlers should run in the order they were registered. As transactions may include multiple

function calls and returns, handler developers should rely only on heap allocated data.

Use: Commit handlers allow us to finalize tasks at transaction commit. System calls with permanent side-effects execute as commit handlers (e.g., `write` to file or `sendmsgs`).

4.2.3 Violation Handlers

Semantics: Violation handlers allow software to register functions to be automatically triggered when conflicts are detected in the current transaction and access information about those conflicts. A transaction can register multiple violation handlers with arbitrary arguments in the *violation handler stack* stored in thread-private memory. The stack base and top are tracked in the TCB fields `xvhptr_base` and `xvhptr_top`, but code located at `xvhcode` is responsible for running all registered handlers (as described below). Violation handlers should run in the reverse order from which they were registered to preserve correct undo semantics.

Like commit handlers, violation handlers start as part of the current transaction and have access to its speculative state. They can safely access thread-private state, but should use open-nested transactions to access shared state.

Handlers may access a list of each of the `xvcount` conflicting addresses through the `xvaddrs` TCB element. Since `xvaddrs` is populated from a hardware register (see Section 4.4.2), only a limited number of address may be presented. If more addresses were detected than could be recorded, the `xvoverflow` bit is set.

The violation handler returns by executing the `xvret` instruction, which jumps to the address in `xvPC`. By manipulating `xvPC` before returning, violation handlers can continue the current transaction (i.e., ignore the conflict), roll back and re-execute, or roll back and run other code. To roll back the transaction, the handler must flush the write-buffer or process the undo log, discard the read-set and write-set using `xrwsetclear`, and restore the register checkpoint with `xregrestore`.

Use: Violation handlers allow for software contention management on a conflict [42, 35] (see Section 2.6). It also allows for compensation code for system calls that execute within the transaction if it rolls back (e.g., `read` or `lseek` to a file).

4.2.4 Abort Handlers

Semantics: Abort handlers are identical to violation handlers but they are triggered when a transaction uses an explicit `xabort` instruction. A separate register points to the code to invoke (`xahcode`). Abort handlers have a separate stack bounded by the `xahptr_base` and `xahptr_top` fields in the TCB. The uses of abort handlers are similar to those of violation handlers.

4.2.5 Nested Transactions

We define two types of nesting, explained in Figure 4.1.

Closed Nesting Semantics: A closed-nested transaction starts when an `xbegin` instruction executes within another transaction (T_B and T_C in Figure 4.1). The HTM system separately tracks the read-set, write-set, and speculative state of the child transaction from that of its parent. However, the child can access any state generated by an ancestor. If a child detects a conflict, we can independently roll back only the child, without affecting any ancestors. When a child commits using `xcommit`, hardware merges its speculative writes (❶) and read-/write-set (❷) with that of its parent, but no update escapes to shared memory. We make writes visible to shared memory only when the outermost transaction commits (❸, ❹).

Open Nesting Semantics: An open-nested transaction starts when an `xbegin_open` instruction executes within another transaction (see T_E in Figure 4.1). Open nesting differs from closed nesting only in commit semantics. On open-nested commit, we allow the child transaction to immediately update shared memory with its speculative writes (❸, ❹). The parent transaction updates the data in its read-set or write-set if they overlap with the write-set of the open-nested transaction. However, conflicts are not reported and no overlapping addresses are removed from the parent's read-set or write-set. If we want to undo the open nested transaction after it commits and its parent aborts, we need to register an abort and/or violation handler.

Our closed nesting semantics are identical to those presented by Moss and Hosking [73]. However, our open nesting semantics differ. Moss and Hosking propose that open-nested transactions remove from their ancestors' read-set or write-set any addresses

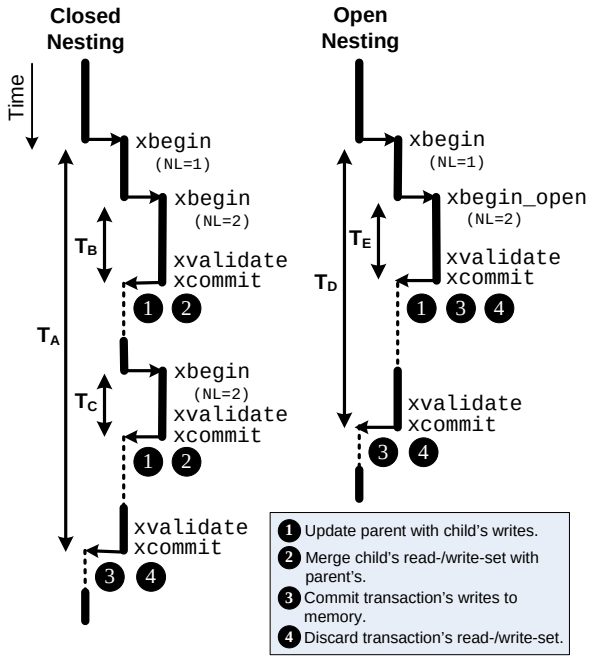

Figure 4.1: Timeline of three nested transactions: two closed-nested and one open-nested.

they update while committing. Their motivation is to use open nesting as an *early release* mechanism that trims the read-/write-set for performance optimizations [30]. We find these semantics non-intuitive and dangerous: an open-nested transaction within library code that is a black box to the programmer can change the atomicity behavior of the user's code in an unanticipated manner.

Unfortunately, there are pitfalls in our open nesting semantics. First, if the open-nested transaction T_D registers a violation/abort handler to be run when its parent, T_E, aborts, that handler's view of memory will be that at the end of T_E instead of at the end of T_D. Some advocate "rolling back" the state of T_E to the proper point before executing T_D's violation/abort handler [70]. Unfortunately, this can only be done in a system with eager versioning.

Also, if sharing is implemented at the cache line level, an open-nested transaction may expose a parent's write: if a parent writes to a line and its open-nested child writes to a different address in the same line, the open-nested commit will expose the entire line.

All these issues are examples of a more general problem: what should be the semantics of open-nested children that access their parent's data? Agarwal et al. [4] describes these problems, including serializability and composability issues. Both Agarwal et al. and Moravan et al. [70] observe that these problems are eliminated if children and their compensating actions do not write the same data as any of their ancestors (starting with their immediate parent), an assumption they call O1.

It suffices to say that open nesting should be used by system-level programmers and library designers to implement complicated functionality and not whimsically employed by application programmers looking for increased performance.

Use: Closed-nested transactions allow for independent rollbacks and contention management at each nesting level, which typically leads to better performance. Open-nested transactions can both roll back and commit independently from their parents, which provides a powerful tool for system code development (this was independently observed by Moss et al. [72]). We can use them within a transaction to perform system calls without creating frequent conflicts through system state (e.g., `time`). We can use them for calls that update system state before parents commit (e.g., `brk`). We also use them in handlers to access shared state independently. Note that within an open-nested transaction, we still provide atomicity and isolation, hence system code does not need locks for synchronization. In many cases, open-nested transactions must be combined with violation handlers to provide undo capabilities.

4.2.6 Nested Transactions and Handlers

Nested transactions can have separate handlers. To properly track all information, each transaction has its own TCB frame. We implement a stack of TCB frames in thread-private memory as shown in Figure 4.2, with a frame allocated before `xbegin` or `xbegin_open` and deallocated on `xcommit` or a rollback. The base and current top of the TCB stack are identified by registers `xtcbptr_base` and `xtcbptr_top`. TCB frames have fixed length as the read-set, write-set, and speculative state of the transaction are physically tracked in caches or undo logs. For each transaction, `xstatus` tracks the current nesting level. Overall, TCB management for transactions is similar to activation record

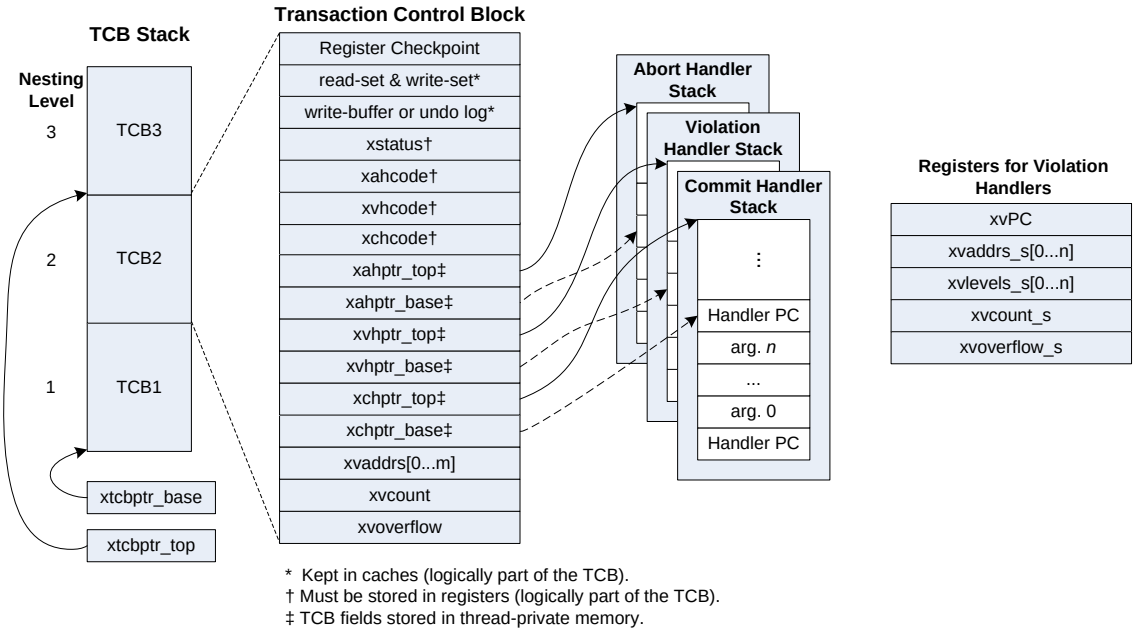

Figure 4.2: The Transaction Stack containing three Transaction Control Blocks (TCBs), one per active nested transaction. The second entry is shown in detail, complete with commit, violation, and abort handler stacks. Also shown are the additional registers needed for violation handlers.

management for function calls.

A single stack is necessary to store all registered handlers of a certain type. Each transaction has separate base and top pointers to identify its entries in the stack. At commit, a closed-nested transaction merges its commit, violation, and abort handlers with those of its parent by copying its top pointer (e.g., `xchptr_top`) into the parent's top pointer. The fixed length of TCB frames makes such an operation trivial. On an open-nested commit, we execute commit handlers immediately and discard violation and abort handlers. If the violation/abort handlers need to persist, the open-nested commit handler should copy them to the parent's violation/abort handler stack.

With nesting, conflicts can be detected for a transaction at any active nesting level and some conflicts may affect multiple levels at once. We always run the violation handler of the innermost transaction (top TCB), even if the conflict involves one of its parents. This is convenient as it allows software to run violation handlers at all levels as needed. It is also required for open-nested transactions that execute system code, as system handlers should be the first to be invoked.

4.2.7 Non-Transactional Loads and Stores

Semantics: An immediate load (`imld`) does not add the address to the current transaction read-set. An immediate store (`imst`) updates memory immediately without a commit and does not add the address to the current write-set. We also introduce an idempotent immediate store (`imstid`) that does not maintain undo information for the store in the writebuffer or the undo log either. These immediate accesses can be interleaved with regular accesses tracked by HTM mechanisms—i.e., there is no non-transactional "region," but rather single instructions.

Use: Transactions often access thread-private data such as various fields in their TCB. To reduce the pressure on HTM mechanisms on such accesses, immediate loads and stores can be used. However, these should only be used when the compiler or system developer can prove they access thread-private or read-only data. Other useful places to use non-transactional accesses may be in handlers, where generating additional dependencies is not desired.

4.2.8 Discussion

Some advocate providing an early release mechanism [30], including the original published version of this work [65], which removes an address from the transaction's read-set. This instruction is attractive for performance tuning, but can complicate programming. Currently, we do not advocate its use in a high-level programming language and believe it can also be excluded from the ISA [96]. Early release is difficult to implement consistently in some HTM systems: if the read-set is tracked at cache-line granularity, and an early release instruction provides a word address, it is not safe to release the entire cache line.

We do not support mechanisms to temporarily *pause* or *escape* a transaction and run non-transactional code. While, such mechanisms may seem attractive for invoking system calls, we find them redundant and dangerous. Open-nested transactions allow us to run (system) code independently of the atomicity of the currently running user transaction. Moreover, open-nesting provides independent atomicity and isolation for the system code as well. With pausing or escaping, a system programmer would have to use lock-based synchronization and deal with all its shortcomings (deadlocks, races, etc.). We believe the benefits of TM synchronization should be pervasive even in system code.

A recently proposed half-way point between open nesting and pause/escape regions for implementing non-transactional operations (like I/O) is called irrevocable transactions [103]. Before a transaction performs I/O, an `irrevocable` instruction is executed which permanently validates the transaction, preventing it from rolling back even in the face of future conflicts. This is similar to but fundamentally different than simply executing `xvalidate`—after `xvalidate`, new dependencies can be created and new conflicts detected; after `irrevocable`, no dependencies are generated. We believe that open nesting is sufficient to implement most inherently non-transactional operations, but `irrevocable` could be useful when interacting with devices that, for whatever reason, cannot have their I/O buffered.

Language	Nesting	Two-phase/Commit Handlers	Violation & Abort Handlers
Harris et al. [41, 40]	open for `AtomicAbortException`; cond. synch.	I/O	undo I/O; exceptions; cond. synch.
Welc et al. [102]	–	–	–
Harris et al. [42]	closed; open for cond. synch.	–	cond. synch.
AtomCaml [83]	open for cond. synch.	–	cond. synch.
Atomos [17]	closed; open for cond. synch.	I/O	cond. synch.
X10 [22]	open for cond. synch.	–	cond. synch.
Chapel [30]	–	–	–
Fortress [6]	–	–	`tryatomic`

Table 4.3: The HTM mechanisms needed to implement recently proposed transactional programming languages. We list which language features are implemented by each mechanism. "–" means this feature is not needed.

4.3 Flexibly Building Languages and Systems

Section 4.2 generally described the uses of the proposed HTM mechanisms. This section provides specific examples that implement language or system code functionality to showcase the expressiveness of the mechanisms. We argue that these mechanisms provide all or most of the flexibility needed to implement reasonable TM systems.

Transactional Programming Languages: We studied the proposed languages for programming with TM including Harris et al. [41, 40], Welc et al. [102], Transactional Haskell [42], X10 [22], Chapel [30], Fortress [6], AtomCaml [83], and Atomos [17]. The proposed ISA semantics are sufficient to implement these languages on HTM systems. Some languages formally support closed nesting [42, 17], and Atomos supports open nesting [17]. Additionally, open nesting can be used to implement the `AbortException` construct in Harris [40], conditional synchronization [42, 22, 83, 17], transactional collection classes [16], and transactional I/O [42, 17]. Two-phase commit and commit handlers are used for I/O and transactional collection classes as well. Violation and abort handlers are used for error handling [42], the `tryatomic` construct in X10 [22], and in most implementations of conditional synchronization and I/O. Table 4.3 summarizes which semantic components are required for which language functionality.

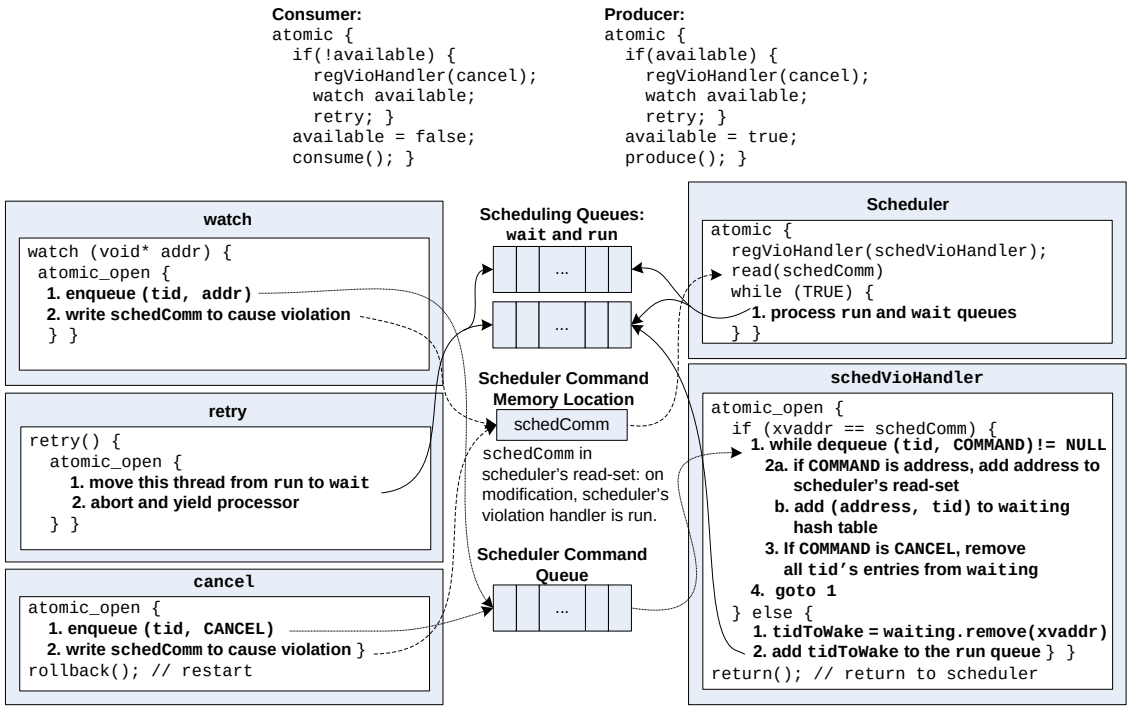

Figure 4.3: Conditional synchronization using open nesting and violation handlers for producer/consumer code in the Atomos programming language [17].

Conditional Synchronization: Figure 4.3 illustrates the concept of conditional synchronization for producer/consumer code in the Atomos programming language [17]. When a transaction wishes to wait for a value to change, it adds the corresponding addresses to a *watch-set* and yields the processor. If any values in the watch-set change, the thread is re-scheduled. This has the attractive property of avoiding the need for an explicit notify statement: the notifier does not need to know explicitly that someone is waiting for this value as the system automatically detects the change using conflict detection.

We implement this functionality using open nesting and violation handlers. An open-nested transaction communicates a waiting thread's watch-set to a scheduling thread that incorporates it as part of its read-set. Hence, the scheduler will then receive conflicts when any values in the watch-set change. Its violation handler will then add the proper thread to the run queue. To communicate with the scheduler, the waiting thread uses open nesting to write to a command queue and then violates the scheduler via the shared `schedComm` variable. Further details about conditional scheduling in Atomos are available [17]. With the proposed HTM mechanisms, we can implement similar runtime systems for other languages that support conditional synchronization within transactional code [42, 83, 22].

System Calls and I/O: To illustrate the implementation of system calls within transactions, we discuss I/O such as `read` and `write` calls without serialization. For input, we perform the system call immediately but register a violation handler that restores the file position or the data in case of a conflict. The system call itself executes within an open-nested transaction to avoid dependencies through system code. For output, we provide code that temporarily stores data in user buffers and registers a commit handler to perform the actual system call. This transactional scheme works with simple request/reply I/O, often the common case in applications [78, 27]. In a similar manner, we can implement other system calls within transactions that read or write system state. For example, a memory allocator can execute as an open-nested transaction including the `brk` call. For C and C++, a violation handler is registered to free the memory if the transaction aborts. For managed languages like Java and C#, no handler is needed, as garbage collection will eventually deallocate the memory.

Transactional Collection Classes These ISA primitives are also used in the Atomos programming language to implement performance enhancing, yet simple, transactional data-structures [16]. For example, if two transactions are inserting two different values into a binary search tree but the values will be appended to the same leaf node, then the transactions will conflict even though logically, there is no data-structure-level conflict. Using the primitives proposed above, Atomos implements a database concept called Semantic Concurrency Control to prevent memory-level conflicts that are not logical conflicts. Atomos uses open nesting and transactional handlers to store semantic conflict information but defer modification of the data-structure until commit.

4.4 Hardware Implementation

This section summarizes the hardware implementation of the mechanisms presented in Section 4.2. Our goal is to demonstrate that they have practical implementations compatible with current HTM proposals [39, 8, 80, 69].

4.4.1 Two-Phase Commit

The `xvalidate` instruction is a no-op for closed-nested transactions. For outer-most or open-nested transactions, the implementation must guarantee that the transaction cannot violate due to prior memory accesses once `xvalidate` completes. For HTM systems with eager conflict detection, `xvalidate` must block until all previous loads and stores are known to be conflict-free by acquiring exclusive (for stores) or shared (for loads) access to the corresponding data. If timestamps are used for conflict resolution, the conflict algorithm must guarantee that a validated transaction is never violated by an active one even if it has a younger timestamp. For optimistic conflict detection systems, `xvalidate` triggers conflict resolution, which typically involves acquiring ownership of cache lines in the write-set.

The `xcommit` instruction atomically changes the transaction status to committed. Finishing the transaction also involves either resetting the undo log or merging the contents of the writebuffer to shared memory. These steps can be executed within `xcommit`

or in a lazy manner after `xcommit` returns. We discuss the implementation of `xcommit` for nested transactions in Section 4.4.3.

4.4.2 Commit, Violation, and Abort Handlers

The stack management for handlers is done in software without additional hardware support, other than some TCB fields stored in registers (see Table 4.1) and hardware needed for violation information collection (see below). Handlers allow for additional functionality in HTM systems at the cost of additional overhead for commit, violation, or abort events. Since transactions with a few hundred instructions are common [27], our handler registration and management code is based on carefully tuned assembly. The code is also optimized for the common case of a commit without any registered commit handler or a violation that restarts the transaction without any registered violation handler. We present quantitative results in Section 4.5. Note that the same assembly code for handler management can be used by all languages or system code that builds upon the proposed semantics.

Additional hardware is required to invoke violation handlers and to deliver the semantic information like conflict addresses described in Section 4.2.3. To store this information, the hardware appends conflicting addresses to the `xvaddrs_s` register as they are detected, setting `xvoverflow_s` if more addresses have been detected than can be stored. The nesting level of the conflict with address `xvaddrs_s[`i`]` is placed in `xvlevels_s[`i`]`.

The mechanisms for invoking and returning from the violation handler resemble a user-level exception. The hardware interrupts the current transaction by saving its PC in the `xvPC` register and jumps to the code indicated by `xvhcode`. Then, each address `xvaddrs_s[`i`]` is appended to the `xvaddrs` register of the nesting level `xvlevels_s[`i`]`. `xvcount` is used to indicate the next free slot in `xvaddrs`; `xvcount` is incremented for each address appended. Additionally, if a conflict is registered for a level, its `xstatus` is set to *pending abort* if they are not already *aborting*. This is to remember which handlers must be run after a handler finishes. If `xvaddrs` overflows or if `xvoverflow_s` is set, `xvoverflow` is set. Finally, the hardware shadow registers are cleared, but are updated

normally if new conflicts are detected.

To avoid repeated jumps to `xvhcode` if additional conflicts are detected, we automatically disable further violation reporting until either a nested transaction is created (see below) or the handler returns using `xvret`. After a handler returns, the `xstatus` registers of each TCB are checked. If any are *pending abort*, then the violation/abort handler for the topmost transaction waiting to abort is invoked, repeating the process.

4.4.3 Nested Transactions

For nested transactions, the hardware must separately manage the speculative state, read-set, and write-set for each active transaction. On a nested commit, we must merge its speculative state, read-set, and write-set with those of its parent. An additional complication is that multiple transactions in a nest may be writing the same cache line, which requires support for multiple active versions of the same data. While it is attractive to disallow this case, this will overly complicate the development of transparent libraries where arguments and return values can be frequently written by two transactions in a nest.

Hence, nesting requires significant changes to the caches used in HTM systems for read-/write-set tracking and speculative buffering. We propose two basic approaches: (1) the *multi-tracking scheme* that allows each cache line to track the read-set and write-set for multiple transactions, and (2) the *associativity scheme* that uses different lines in the same cache set to track multiple versions of the same data [99]. Even though there are numerous possible combinations of these two schemes with all other HTM design options, we will briefly describe two practical design points that illustrate their implementation details.

Nesting Support with Multi-tracking Lines

We consider the multi-tracking scheme for HTM designs using eager versioning. Each cache line tracks read-/write-set membership for multiple transactions in the nest, but the multiple versions of the data are buffered in the undo log. Each transaction tracks the base point of its entries in the undo log using a separate register or TCB field. On a

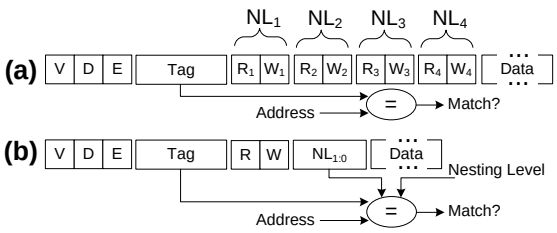

Figure 4.4: Cache line structure for (a) multi-tracking, and (b) associativity schemes with four levels of nesting.

closed-nested commit, the log entries are automatically appended to those of its parent without any action. On a nested conflict, we can roll back by processing the undo log entries for this nesting level in FILO order. The only complication is that if an open-nested commit overwrites data also written by its parent, we must update the log entry of the parent to avoid restoring an incorrect value if the parent is later rolled back. This requires a potentially expensive search through the undo log.

Figure 4.4(a) shows the cache line organization for multi-tracking. A line has multiple copies of the R_i and W_i bits that indicate membership in the read-set and write-set for the transaction at nesting level i (NL_i). If word-level tracking is implemented, we need per-word R and W bits. On a memory access, the hardware uses the nesting level counter in `xstatus` to determine which bits to set. On an cache lookup for conflict detection, all R and W bits are checked in parallel. Hence, we can detect conflicts for all active transactions in parallel. On a rollback at NL_i, we gang invalidate (flash clear) all R_i and W_i bits. On a closed-nested commit at NL_i, we must merge (logical OR) all R_i bits into R_{i-1} and all W_i bits into W_{i-1}. Such merging is difficult to implement as a fast gang operation. To avoid latency proportional to the read-set and write-set size of the nested transaction, we can merge *lazily* while continuing the execution at NL_{i-1}: on a load or store access to a cache line, we perform the merging if needed with a read-modify-write as we lookup the cache line and update its LRU bits. During the lazy merge, the conflict resolution logic should consider R and W bits at both levels as one. The merging must complete before a new transaction at the level NL_i level is started. On an open-nested commit at NL_i, we simply gang invalidate all R_i and W_i bits.

Nesting Support with Associative Caches

We consider the associativity scheme for HTM designs using lazy versioning. As always, the cache tracks read-/write-set membership, but also buffers multiple versions of old data for nested transactions. Figure 4.4(b) shows the new cache line organization. Each line has a single set of R and W bits, but also includes a nesting level field (NL) to specify which transaction it holds data for. We reserve $NL = 0$ for cached data that do not yet belong to any read- or write-set. If the same data is accessed by multiple nested transactions, we use different lines in the same cache set. The most recent version is in the line with the highest NL field. Hence, cache lookups can return multiple hits and additional comparisons are necessary to identify the proper data to return, so cache access latency may increase in highly associative caches. On an access by a transaction at NL_i, we first locate the most recent version. If the cache line has $NL = 0$ (potentially after a cache refill), we change $NL = i$ and set the R or W bits. If there is another speculative version at level $i-1$ or below, we first allocate a new line in the same set that gets a copy of the latest data and uses $NL = i$. On an external lookup for conflict detection, we check all lines in the set with $NL \neq 0$ to detect, in parallel, conflicts at all nesting levels.

On a rollback at NL_i, we must invalidate all cache entries with $NL = i$. This can be implemented as a gang invalidate operation if the NL field uses CAM bits. Alternatively, the invalidation can occur *lazily* as we access cache lines and before we start a new transaction at NL_i. On a closed nested commit at NL_i, we must change all lines with $NL = i$ to $NL = i-1$. If an $NL = i-1$ entry already exists, we merge its read-set and write-set information into the older entry and then discard it. Again, the best way to implement this is lazily. While lazily merging, the conflict detection logic must consider cache lines with $NL = i$ and $NL = i-1$ to belong to the same transaction. An open nested commit is similar, but now we change entries from $NL = i$ to $NL = 0$. If there are more versions of the same data for other active transactions, we also update their data with that of the $NL = i$ entry without changing their R or W bits.

Discussion

Each nesting scheme has different advantages. The multi-tracking scheme does not complicate cache lookups and avoids replication for lines with multiple readers. The associativity scheme scales to a number of nesting levels equal to the total associativity of private caches without significant per-line overhead. A hybrid scheme with multi-tracking in the L1 cache and associativity in the L2 cache may provide the best of both approaches.

Any practical HTM implementation can only support a limited number of nesting levels. Hence, nesting levels must be virtualized just like any other buffering resource in HTM systems. Virtualization schemes like Rajwar et al. [80] can be extended to support unlimited nesting levels by adding a nesting level field in each entry in the overflow tables in virtual memory. Early studies have shown that the common case is 2 to 3 levels of nesting [27], which is easy to support in hardware. The hardware support for nesting can also be used to overlap the execution of independent transactions (double buffering), which can be useful with hiding any commit or abort overheads [66].

Moss and Hosking [73] discuss a nesting implementation similar to our associativity scheme. Apart from the different semantics for open nesting (see Section 4.2.5), their approach is overly complex. In addition to the *NL* field, each line includes a variable length stack with pointers to all child transactions with new versions of the data. To maintain the stacks, we need to push or pop values from multiple cache lines on stores, commits, and aborts. The two schemes we propose are simpler and introduce lower area and timing overheads.

The proposed nesting schemes do not support nested parallelism: the ability to execute a single transaction atomically over multiple processors [62, 6, 3, 44, 71, 76]. 2, 10, 16, 17 Nested parallelism requires that we can merge read-sets and write-sets across processors as parallel tasks complete within a transaction. Such functionality can be implemented using a shared cache between collaborating processors that serves as a directory for the latest version of any data. Note, however, that the semantics of nesting defined in Section 4.2.5 are correct even for nested parallelism.

4.5 Evaluation

The simulation environment used for these experiments is slightly different than that of Chapter 3—principally, the simulator runs PowerPC code instead of x86. Our goal was to examine optimization opportunities and use the new mechanisms to implement the programming constructs and runtime code functionality discussed in Section 4.1. An extensive comparison of alternative implementations for the proposed semantics was beyond the scope of this research.

The simulator uses an LO HTM system and supports three levels of nesting using the associativity scheme with lazy merging. None of the evaluated programs use more than $NL = 2$. The system simulates up to 16 cores, with private L1 Caches (32KB, 1-cycle access), and private L2 caches (512KB, 12-cycle access). The processors communicate over a 16-byte, split-transaction bus.

The simulator models all overheads associated with two-phase commit and the software for TCB and handler management. We carefully optimized the assembly code for common events to avoid large overheads for small transactions. Starting a transaction requires 3 instructions for TCB allocation. A commit without any handlers requires 4 instructions, while a rollback without handlers requires 5 instructions. Registering a handler without arguments takes 3 instructions. Note that some of these instructions access thread-private data in memory and may lead to cache misses. Yet, such misses are rare as private stacks cache well. Overall, the new HTM functionality does not introduce significant overheads.

4.5.1 Performance Optimizations with Nesting

This evaluation used nested transactions to reduce the overhead or frequency of conflicts in scientific and enterprise applications. We used `swim` and `tomcatv` from the SPECcpu2000 suite [97]; `barnes`, `fmm`, `mp3d`, and `water-nsquared` (called simply `water`) from the SPLASH and SPLASH-2 suites [94, 104]; and a C version of `moldyn` from the Java Grande suite [55]. For these applications, we used transactions to speculatively parallelize loops. We also used a modified version of `SPECjbb2000` [98] running on the Jikes RVM [7]. Note that these are slightly different parallelizations with different transaction

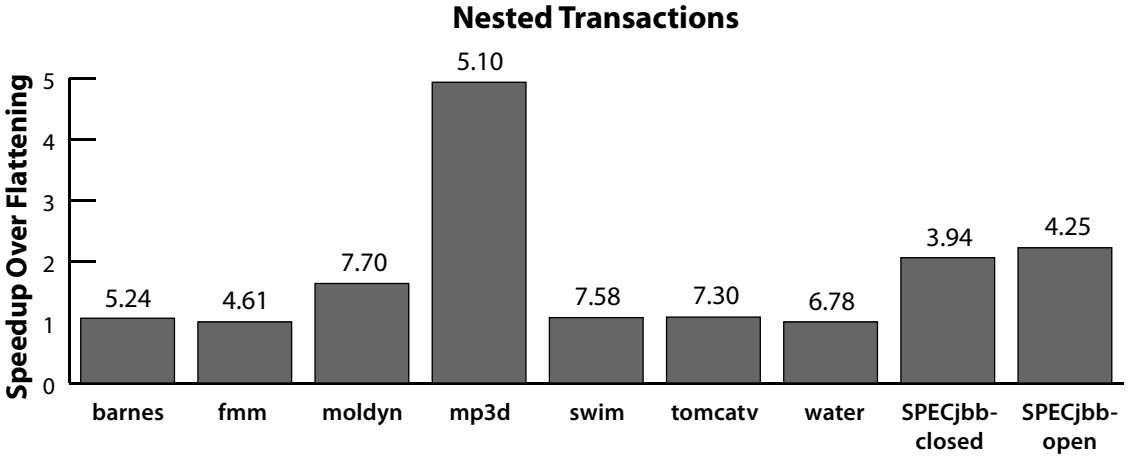

Figure 4.5: Performance improvement with full nesting support over flattening for 8 processors. Values shown above each bar are speedups of nested versions over sequential execution with one processor.

boundaries than those of the same applications in Chapter 3.

Figure 4.5 plots the performance improvement achieved with the proposed nesting implementation over the conventional HTM approach that simply flattens nested transactions. The results were produced running 8 processors. The number above each bar reports the overall speedup achieved with nesting over sequential execution on one processor (maximum speedup is 8). Overall, no application is affected negatively by the overhead of TCB and handler management for nested transactions. Most outer transactions are long and can amortize the short overheads of the new functionality. Most inner transactions are short, hence lazy merging at commit does not become a bottleneck. On the other hand, several applications benefit significantly from the reduced cost of conflicts compared to flattening. For the scientific benchmarks, we applied closed nesting mainly to update reduction variables within larger transactions. This allows us to avoid several outer transaction rollbacks, particularly when the inner transaction is near the end of the outer one. The improvements are dramatic for `mp3d` (4.93×) where we also used nesting to eliminate expensive violations due to particle updates on collisions.

As a three-tier enterprise benchmark, `SPECjbb2000` is far more interesting from the point of view of concurrency. We parallelized `SPECjbb2000` within a single warehouse,

where customer tasks such as placing new orders, making payments, and checking status manipulate shared data-structures (B-trees) that maintain customer, order, and stock information. Conceptually, there is significant concurrency within a single warehouse, as different customers operate mostly on different objects. Nevertheless, conflicts are possible and are difficult to statically predict. We defined one outer-most transaction for each `SPECjbb2000` operation (order, payment, etc.). Even though we achieved a speedup of 1.92 using this flat-transaction approach, rollbacks significantly degrade performance. With nesting support, we developed two additional versions of the code. The first version, `SPECjbb2000-closed`, uses closed-nested transactions to surround searches and updates to B-trees. Performance is improved by 2.05× (total speedup of 3.94) as the violations occur frequently within the small inner transactions and do not cause the outer-most operation to roll back. The second version, `SPECjbb2000-open`, uses an open-nested transaction to generate a unique global order ID for new order operations. Without open nesting, all new order tasks executing in parallel will experience conflicts on the global order counter. With open nesting, we observe a performance improvement of 2.22× (total speedup of 4.25) as new orders can commit the counter value independently before they complete the rest of their work. Hence, conflicts are less frequent and less expensive. Note that no compensation code is needed for the open-nested transaction as the order IDs must be unique, but not necessarily sequential. We could use both open and closed nesting to obtain the advantages of both approaches, but we did not evaluate this.

4.5.2 I/O within Transactions

We designed a C microbenchmark where each thread repeatedly performs a small computation within a transaction and outputs a message into a log. We designed a transactional library function that buffers output in a private buffer and registers a commit handler before returning control to the application. If the transaction violates, the local buffer is automatically discarded because it is part of the write-set. If the transaction successfully validates, the commit handler copies the local buffer to a shared buffer in the operating system.

We evaluated three versions of this I/O microbenchmark: the transactional one described above and two using conventional lock-based synchronization. The coarse-grain version acquires a lock at the boundaries of the library function call; so it formats and copies the log message into the shared buffer while holding the lock. The fine-grain version holds the lock only while copying to the shared buffer. Figure 4.6 compares the performance of the three versions as we scale from 1 to 16 processors. Since there is little work per repetition except printing to the log, the coarse-grain version completely serializes and shows no scalability. The fine-grain version allows for concurrency in message formatting. Still, its scalability is limited by serialization on buffer updates and the overhead of acquiring and releasing the lock. Initially, the performance of the transactional version suffers because it performs two copies: one to a local buffer then one to the shared buffer. However, because the overhead is fixed and does not scale with the number of threads, as the lock overhead does, the transactional version continues to scale. With more processors, we would notice the HTM version flattening as well due to conflicts when updating the shared buffer.

Despite its simplicity, the I/O benchmark shows that the proposed semantics allow for I/O calls within transactions. Moreover, despite the overhead of extra copying, transactional I/O in a parallel system scales well.

4.5.3 Conditional Synchronization within Transactions

To measure the effectiveness of the conditional synchronization scheme in Figure 4.3, we use the Atomos `TestWait` microbenchmark, which mimics an experience from Harris and Frasier [41], that stresses thread scheduling in a producer-consumer scenario. `TestWait` creates 32 threads, arranged in a ring of producers and consumers with a shared buffer between each pair. The benchmark scales the number of tokens in the ring and begins passing a fixed number of tokens from one thread to the next. Every thread uses a transaction to perform one token operation (dequeue from receiving buffer and copy to the outgoing buffer). An efficient system should scale with the number of tokens.

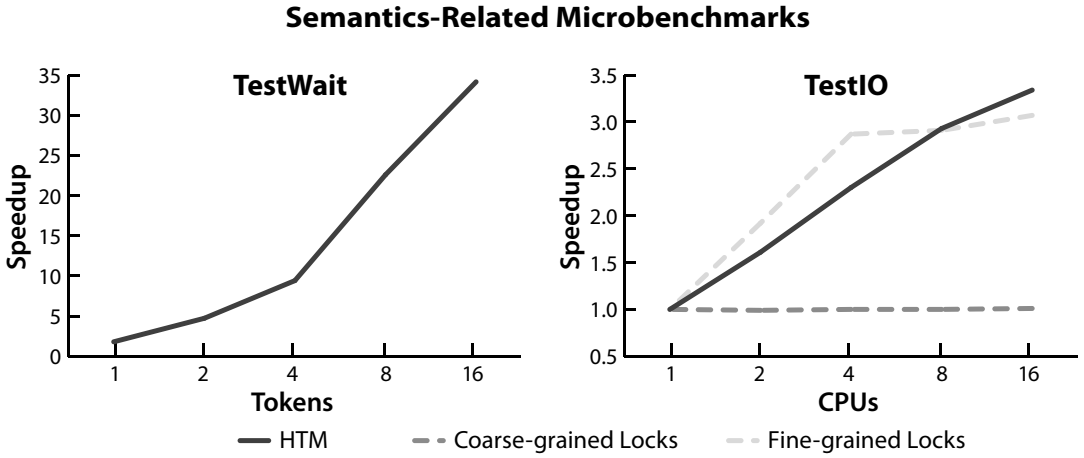

Figure 4.6: On the left, speedup of token exchanges for `TestWait` with 32 processors, scaling the number of tokens from 1 to 16. On the right, speedup for the I/O microbenchmark with HTM, fine-grain locks, and coarse-grain locks.

Figure 4.6 shows that passing the tokens scales super-linearly with the number of tokens in the ring; this is expected since with more tokens, a consumer may find an available token immediately without having to synchronize with the producer. See further discussion in the evaluation of the Atomos programming language [17]. Hence, conditional synchronization within transactions using open nesting and violation handlers is quite effective for the studied system. Both the open-nested transactions and the violation handlers are quite small and introduce negligible overhead.

4.6 Conclusion

For HTM systems to become useful to programmers and achieve widespread acceptance, we need rich semantics at the instruction set level that support modern programming languages and system code. I proposed the first comprehensive ISA for HTM that includes three key mechanisms (two-phase commit, transactional handlers, and open- and closed-nested transactions). I described the hardware and software conventions necessary to implement these mechanisms in various HTM systems. Moreover, I have quantitatively evaluated the proposed mechanisms by showing their use for both system

code functionality (scalable I/O and conditional synchronization) and performance optimizations through nested transactions.

Armed with these implementation-independent semantics at the instruction set level, the TM community can effectively develop and evaluate hardware proposals that are practical for programmers and support a wide range of applications. Similarly, software researchers can design efficient languages and runtime systems for HTM on top of a rich interface between their programs and the underlying hardware.

Chapter 5

All Transactions All the Time

In previous chapters, we examined systems that supported transactions specified by users and executed non-transactional code using some other coherence and consistency mechanism. It is conceivable and indeed beneficial to consider a system that not only executes code within programmer-defined `atomic` regions as transactions, but also executes all other code inside some form of transaction. In such a continuously transactional system, there is only one execution mode (transactions) and as such, all thread communication, including that required to maintain coherence and consistency, is achieved through transactional commit. This chapter explores the potential advantages of "all transactions all the time" and evaluates an implementation called Transactional Coherence and Consistency.

Continuously transactional systems provide a number of advantages to both hardware and software developers:

- **Hardware Complexity and Scalability.** If coherence and consistency need only be maintained at transaction boundaries, then conventional protocols for these concepts can be discarded. Furthermore, this larger scale communication creates less sensitivity to latency (vis a vis MESI coherence messages) and more exploitation of available bandwidth (since write-set data can be aggregated), creating better scalability.

- **Software Programmability.** Continuous transactions provide programmers with

a single, high-level abstraction to reason about parallelism, communication, consistency, and failure atomicity, making for a more intuitive programming model. Also, with transactions offering a larger observable unit than instructions, it is easier to monitor program progress, creating opportunities for low-overhead collection of profiling and debugging data. Finally, when applied in a continuous way, TM constructs prove useful for creating various tools like deterministic replay of parallel systems, traditionally a hard problem.

To build a system that provides these benefits , we introduce Transactional Coherence and Consistency (TCC), built by modifying LO. This chapter provides the following contributions:

- **Motivation and Implementation.** We motivate and describe an efficient implementation of a continuously transactional HTM system called Transactional Coherence and Consistency, which is an extension of Lazy-Optimistic, described in Section 2.3.

- **Performance Comparison on Multicores.** We provide the first execution-driven simulation results of a continuously transactional HTM. A straight-forward implementation of TCC scales and performs adequately compared to its ancestor, LO.

- **Design Alternatives.** We evaluate three TCC design alternatives: choice of coherence granularity (word or cache line) for speculative state tracking, snooping protocol (invalidate or update), and choice of commit protocol (commit-through or commit-back). Experiments show performance is not significantly impacted by snooping protocol or commit protocol, though using a word-level coherence granularity clearly benefits applications with fine-grained sharing patterns and false sharing.

- **Interconnect Utilization.** Continuous transactions means more transactions to commit, possibly stretching available interconnect bandwidth. However, the commit bandwidth requirements for TCC are only slightly higher than with other TM schemes, but well within the capabilities of bus-based interconnects, even for a multicore processor with 32 CPUs.

- **Buffering Requirements.** Efficient continuous transactional execution does not require more buffering than the transactional systems presented in Chapter 2 when combined with a periodic commit scheme (see Section 5.1). Similar to those systems, TCC uses private caches to buffer speculative state and simple victim caches to eliminate associativity overflows.

Our results show that continuous transactional execution is practical to implement for multicore systems. TCC provides for easier parallel programming without significant compromises in the peak performance possible for each application.

In fact, after our work on TCC, other projects have adopted continuous transactions. The Bulk Multicore Architecture [100], envisioned by researchers at UIUC, is meant to be a general-purpose, easy-to-program architecture supporting advanced programming and debugging tools. It accomplishes this by continuously executing "chunks," which are very similar to transactions: groups of instructions executing atomically. Also, a team at Microsoft Research has developed Automatic Mutual Exclusion [54], a programming model built around continuous transactions to provide the programming advantages listed earlier in this introduction.

5.1 Continuous Transactional Execution

Continuous transactions means executing transactions at all times—every instruction in the system is part of some transaction. Obviously, every programmer-defined transaction is mapped to a single hardware transaction, but even code between `atomic` regions is run as one or more transactions. This creates two types of transactions: programmer-defined transactions we call "explicit" and other transactions we call "implicit."

Both implicit and explicit transactions have the same basic semantics as the transactions we encountered in previous chapters: they are executed atomically and isolated from other transactions. As such, they can be implemented using similar techniques as those described in Chapter 2. However, the hardware in those TM implementations was also required to execute non-transactional threads, requiring the appropriate logic and communication to maintain coherence and consistency between those threads. Continuous transactions allows us to replace coherence and consistency protocols with only

transactions, greatly simplifying both the programmer's view of communication and the hardware implementation.

Traditional consistency protocols use special instructions and messages to reason about and maintain order between individual memory references. A continuously transactional model maintains no such ordering but instead imposes a serializable ordering between entire transactions. From a memory model point of view, all memory references contained within transactions that committed earlier happened before all memory references contained within transactions that committed after, regardless of the actual interleaving of the accesses. In fact, we can implement relaxed memory models within transactions, making more compiler optimizations available, while exposing Sequential Consistency (SC) to the programmer. It is known that Sequential Consistency is easier to reason about [2] and with software correctness tools often assuming SC, a continuously transactional system can exploit those as well [18].

Similarly, traditional coherence protocols require exchanging frequent messages regarding single cache lines to ensure a coherent view of memory. Since transactions ensure the proper ordering and sharing of data between threads via conflict detection, a continuously transactional system maintains coherence without special messages or protocol states. Furthermore, maintaining coherence at this larger granularity (i.e., transactions) this makes it possible to detect, and perhaps even execute correctly, unwanted data races even in code not marked as a transaction by programmers.

Because continuous transactions implement coherence and consistency at transaction boundaries with only basic TM mechanisms, hardware designs can be much simpler. No longer needed are coherence protocols or carefully reasoning about the implementation of memory barriers. And, because transaction commit communicates larger amounts of data more infrequently than single coherence messages, a continuously transactional system is less latency sensitive and stands to better utilize available bandwidth, potentially reducing hardware costs.

Continuous transactions can be also used to implement useful debugging tools. Part

of our Transactional Application Profiling Environment (TAPE) [20] enables programmers to use continuous transactions to easily identify data races with very little additional overhead. Deterministic replay of parallel systems [68] is also possible with continuous transactions. Data race detection and deterministic replay have been implemented before, but hardware support for continuous transactions makes these tools run at "production speed," perhaps allowing them to be "on" all the time.

Unfortunately, a hardware system cannot execute implicit transactions of any size or duration (see Section 2.7). Specifically, implicit transactions are still subject to resource overflow and so must be split. Two methods exist: commit and create new implicit transactions every n cycles [18] or when buffering is expended. For our implementation of TCC, we chose to commit when speculative buffering was expended, maximizing the size of implicit transactions and minimizing the commit overhead.

5.2 Transactional Coherence and Consistency

Unlike previous proposals using transactional memory merely for non-blocking synchronization, TCC uses continuous transactions as the basic unit of parallel work, synchronization, memory coherence, and consistency. In this section, we begin with a simple HTM design from Chapter 2 and modify it to include continuous transactions. We then discuss design alternatives in Section 5.3

To gain the benefits of maintaining coherence and consistency at transaction boundaries, TCC only communicates at transaction boundaries, making the choice of optimistic conflict detection obvious. Therefore, we decided to build TCC on the Lazy-Optimistic (LO) system (see Section 2.3). Beginning with LO, we remove the MESI-based coherence protocol and instead have transactions broadcast their shared data on commit. Other processors snoop these broadcasts to maintain coherence and detect violations. Unlike MESI coherence messages containing only addresses, both addresses and data are sent on a TCC commit.

Versioning: TCC uses a similar versioning scheme as LO, but without the need for MESI transitions. In fact, a simple TCC implementation would require only three bits of state per cache line: valid (V), read (R), and written (W). When a transaction reads a line, it is

marked R, and when a transaction writes a line, it is marked W, just like in LO. There is no requirement for a dirty bit, since when a transaction commits, it sends its write-set data to memory, clears its R/W bits and leaves the line simply V. There is also no requirement for read-exclusive requests, just as within an explicit transaction in LO.

As an optimization, we also include a per-line Exclusive (E) bit to enable cache-to-cache transfers. Recently committed data and lines only present in one cache are marked E, then subsequent snooped reads clear the E bit. This is not a dirty bit since the line can be evicted without writing back because the commit process has already sent the data to memory.

Conflict Detection: TCC uses broadcast packets full of write-set addresses to detect conflicts at commit time, just as LO does: if a cache has a line R or W and detects a commit to the same line, it must signal a conflict.

Validation and Commit: Since we modified our LO design to support TCC, we are using the same commit token and bulk commit packet described in Section 3.2. Of course, with no dirty bits or other writeback mechanism, the TCC commit packet must include data, not only addresses. This data is then snooped by the shared cache and stored there. Because of this "commit-through" policy (analogous to write-through caches), TCC should require more bandwidth than LO and in Section 5.4.6 we evaluate the bus utilization of TCC.

Coherence: Coherence is maintained via the conflict detection and resolution mechanisms. When a processor snoops a commit packet containing a write to a line within its cache, that line is invalidated and a conflict is signaled if necessary. In this way, any transaction reaching commit has read only the most recent version of any line. For our baseline system, we implemented this invalidate protocol, but we also explore and evaluate an update protocol in Section 5.3.2.

Consistency: Transactions provide a serialized ordering between themselves via validation and conflict detection, ensuring that programmers can reason about the ordering between any two accesses. In this way, commits act like memory barriers: the accesses from all committed transactions occurred "before" the accesses of the currently committing transaction and all subsequent transactions' accesses happened "after."

Contention Management: We use the Aggressive policy, just as we did with LO: committing transactions always win conflicts and other transactions always rollback. This guarantees forward progress and since we're using lazy versioning, aborts are efficient.
Virtualization: We do not expect to experience overflow with TCC since our cache configuration is the same as our experiments with LO. However, we handle overflow of explicit transactions like LO does: grab a global token and commit, holding the token until we encounter an explicit commit. This degrades performance by preventing other transactions from committing, but will suffice as a correctness measure. The techniques listed in Section 2.7 for virtualizing LO systems can be used with TCC as well.
I/O: Because TCC is continuously transactional, all I/O in a program is done within transactions. As such, designers of a TCC system must employ one of the I/O strategies outlined in Chapter 4. Fortunately, any of the strategies compatible with LO are also compatible with TCC.

5.3 Design Alternatives

A number of alternatives can be employed to modify the baseline TCC design. This section describes them and their potential advantages and disadvantages.

5.3.1 Coherence Granularity

The baseline design uses cache lines as its granularity of coherence and conflict detection. It has been known for some time that, for traditional machines, this is a fair tradeoff between false sharing between threads and excessive communication between processors [45]. However, since continuous transaction systems communicate only along transaction boundaries, this aggregate communication makes it possible to efficiently track conflicts at word level.

TCC-WORD is a modification of TCC-BASE that uses word-level versioning and conflict detection. It replaces the per-line Valid, R, and W bits with per-word Valid, R, and W bits. On commit, transactions send not only the addresses and data of speculatively written lines, but also a mask representing which words were written, and hence which

words in the commit packet are valid. Processors then use this information for conflict detection and also to invalidate written words.

State transitions work much like they do in line-level granularity: W is set when a word is written and R is set when a word is read. This implements privatization, allowing multiple concurrent writers to the same word. Of course, corresponding changes must be made to the conflict detection logic: a write-write conflict (two words with only their W bits set) is no longer an atomicity violation. With this scheme, an optimization is available: the R bit does not need to be set on a word-sized read if the W bit is already set. A note about sub-word accesses: sub-word reads and writes set the R and W bits, respectively, but to avoid losing updates, sub-word writes must also set the R bit to prevent two transactions writing different bytes in the same word and, at commit time, that being mis-identified as a W-W conflict and wrongly ignored.

Of course, cache miss handling is made somewhat more complex as a line can be in the cache but the requested word can be invalid. The entire line must then be fetched from memory, with care to void overwriting any modified words in the line. Also, a cache-to-cache transfer can only be initiated if a processor has all words in the line; otherwise, the request must go to memory. This may lead to decreased performance in applications with false sharing or high contention. A system could be devised that delivered partial lines, using a per-word Exclusive and Dirty bit, aggregating them in the requesting processor, but we did not implement such a system.

TCC-WORD should perform better than TCC-BASE in cases where false sharing exists between transactions—i.e., where two transactions share the same line but not the same words. False sharing between transactions in TCC-BASE requires one of the transactions to abort, whereas those transactions would not abort in TCC-WORD.

5.3.2 Coherence Protocol

In a traditional system, updating shared cache lines written by other processors is called using an update protocol. Disregarding the additional communication it requires, update should improve performance by reducing cache misses caused by an invalidate

protocol, however the additional communication often results in a net loss in performance [45]. For this reason, our baseline TCC design used an invalidate protocol.

In TCC however, we are already paying the additional communication cost so it is logical to evaluate TCC with an update protocol in a system we call TCC-UPDATE. For TCC-UPDATE, we chose to use word-level granularity checking. When snooping other transactions' commits, instead of invalidating written words, TCC-UPDATE replaces the written word with the new version included in the commit packet.

TCC-UPDATE should perform better than TCC-WORD for applications where transactions frequently read-modify-write shared words. In TCC-WORD, a transaction may read a word, then detect a conflict, invalidate it and restart, only having to refill the word immediately. TCC-UPDATE will still incur an abort, but upon restart the line will already be valid in the cache.

5.3.3 Commit Protocol

We call TCC-BASE's method of committing "commit-through" since it sends its write-set across the interconnect, similar to so-called write-through caches. One could imagine imagine a "commit-back" system where commit data is left dirty in the committing processor and only the addresses are sent across the interconnect, much like LO. We call such a system TCC-BACK.

TCC-BACK requires adding a Dirty (D) bit, similar to the one in MESI. When a line is committed, the D bit is set but the data for the line is not sent in the commit packet, only the address. The dirty data is written back if the line is evicted or transferred (and the D bit cleared) if the cache snoops a load request. To summarize, a cache line may now be in one of the following states: invalid (V=0, R=?, W=?, E=?, D=?), valid (V=1, R=0, W=0, E=0/1, D=0), speculatively read (V=1, R=1, W=0, E=0/1, D=0), speculatively written (V=1, R=0, W=1, E=0, D=1), speculatively read and subsequently written (V=1, R=1, W=1, E=0/1, D=1), and dirty (V=1, R=0, W=0, E=1, D=1). Of course, TCC-BACK must use an invalidate protocol, but could use a word or line granularity coherence protocol; we evaluated only a line-based system. A word-based system would require dirty bits for each word and aggregating multiple dirty words worth of data during cache-to-cache

transfers, complicating the protocol.

TCC-BACK may have an number of advantages. Since data is only sent when and where it is needed by other threads, commit-back should not require as much bandwidth as commit-through. Additionally, since in large-scale systems, the interconnect often consumes a great deal of energy, this bandwidth savings may save energy and therefore, total cost of ownership. Finally, since commit-through requires storing commit data into the shared cache (or memory if no shared cache is present), commit-back should relieve pressure on this shared resource.

5.4 Performance Evaluation

To explore the possibilities of continuous transactional execution, we compared the performance of our TCC implementation to that of LO. Additionally, we implemented and evaluated the various design alternatives discussed in the previous section. This section describes our methodology and results. Note that Table 5.2 presents the speedups of the STAMP applications on LO and all our TCC design variants for easy comparison.

5.4.1 Methodology

We use the same experimental setup and simulator defaults as we did in Chapter 3. Namely, we use the same interconnect, CPU architecture, and cache parameters described in Table 3.1. We chose the Aggressive contention management policy, which is the same as that for LO.

We evaluated our TCC systems using the same STAMP applications described in Chapter 3. We did not change the parallelization of the applications or the location and size of explicit transactions. However, since TCC uses continuous transactions, many new implicit transactions were created, so we present new transactional application characteristics in Table 5.1. Because the statistics now include both implicit and explicit transactions, and because implicit transactions are by definition of a medium size, almost all the applications experience regression toward the mean of more moderately

	PER TRANSACTION								
	instructions*		reads*†		writes*†		writes† per 1K ins*		retries‡
bayes	29,795	(37)	75	(6)	45	(3)	1.5	(81.1)	0.34
genome	846	(78)	26	(9)	7	(4)	7.9	(51.3)	0.09
intruder	610	(71)	14	(9)	6	(6)	9.5	(84.5)	1.04
kmeans	924	(143)	23	(15)	5	(5)	5.3	(35.0)	0.05
labyrinth	98,586	(56)	58	(7)	99	(5)	1.0	(89.3)	0.39
ssca2	140	(50)	12	(9)	6	(4)	41.1	(80.0)	0.01
vacation	1,640	(907)	53	(29)	15	(11)	9.4	(12.1)	0.21
yada	4,700	(35)	26	(4)	14	(3)	2.9	(85.7)	0.25

Table 5.1: Cache and transactional statistics for benchmark applications under continuous transactions. Includes both implicit and explicit transactions. *Averages are presented, with medians in parentheses. †Reads and writes in cache lines. ‡Retries per transaction presented for 16 CPU, TCC-BASE case. Simulation parameters are the defaults presented in Table 3.1.

sized transactions, with a corresponding change in reads/writes per transactions and retries per transaction. For example, the average transaction size in `bayes` and `labyrinth`, with their large explicit transactions, shrinks due to smaller implicit transactions. Similar but smaller changes occur in `genome`, `vacation`, and `yada`. But, `intruder`, `kmeans`, and `ssca2`, with their small explicit transactions, experience increases.

5.4.2 Baseline Evaluation

We begin our evaluation of continuous transactions and TCC by comparing the baseline system to LO-BASE. Execution time breakdowns are presented in Figure 5.1. In general, the performance of most applications is worse than that of LO due to false sharing. Fortunately, it does not appear that performance is significantly affected by using the same level of speculative buffering as LO. Detailed results follow.

bayes

In `bayes` we begin to see a common pattern in the execution time breakdowns between LO and TCC: idle time replaced with time spent on conflicts (Violated time). This

occurs because the loop within barriers is now being run within a transaction, and so when other threads notify the waiting threads by writing a flag, the waiting transactions rollback, marking their time as Violated. This is more of a bug in reporting than an important phenomenon of TCC.

A more important pattern we see is more Violated time, especially at higher processor counts, even if we factor out the time spent in barriers. This is because of false sharing created between transactions: the non-transactional regions in LO can interleave reads and writes to different parts of the same cache line without the need to perform the entire enclosing computation again. A continuous transaction system using line-level granularity cannot differentiate this false sharing from true accesses to the same word, so one of the transactions needlessly aborts.

A final common pattern is increased time spent waiting for refills (Memory and Violate Stall time). This is again due to false sharing and TCC-BASE's invalidate protocol: when a transaction restarts after aborting due to false sharing, it must refill the conflicting line. This increases Violate Stall time when the subsequent transaction aborts and Memory time when the transaction commits.

genome

We see similar patterns in `genome` as in `bayes`. We see Idle/Synch time being replaced with Violated time, more violations due to false sharing, and more refill time. Additionally, since continuous transactions means more simultaneous transactions, there is more contention for commit permission (Validate) time as CPU counts increase.

intruder

`intruders` does not seem to be as affected by false sharing as other applications, as evidenced by its similar performance to LO. However, it does experience some false sharing, as can be seen by the increase in Violate and Violate Stall time.

kmeans

`kmeans` experiences false sharing culminating in more conflicts as CPU counts increase. Also, `kmeans` begins to experience higher Validate time because of the many new transactions introduced in by TCC.

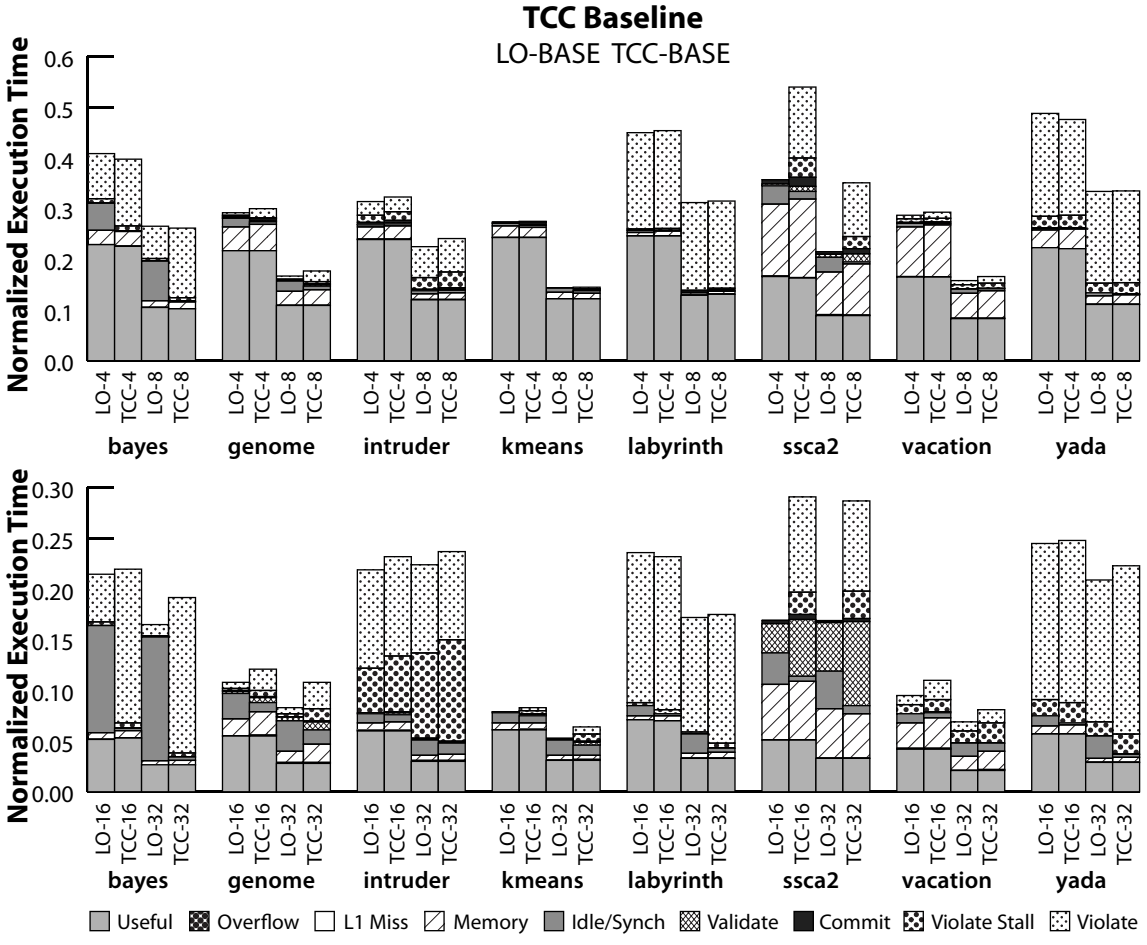

Figure 5.1: Execution time breakdown of STAMP applications on 4–32 CPUs on TCC-BASE compared with LO-BASE. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

labyrinth

`labyrinth`'s performance is not as affected by false sharing as other applications due to the fact that most time is spent in explicit transactions. We do see the characteristic replacement of Idle time by Violate/Violate Stall due to barriers.

ssca2

`ssca2` performs the worst of all applications, partly due to the exacerbation of the SERIALIZEDCOMMIT pathology (more Validate time) we observed in Chapter 3, but also due to false sharing. `ssca2`'s false sharing is particularly bad, as evidenced by the large Violated time, considering almost no true conflicts occur between explicit transactions (see Table 3.2).

vacation

`vacation` on TCC-BASE mostly suffers from the increased memory pressure created by more transactions committing and an already high global L2 miss rate. It also has some false sharing. Both these effects can be seen in the Memory and Violate Stall categories.

yada

`yada`'s story is similar to that of `labyrinth` because it spends most of its time in explicit transactions, minimizing the effect of false sharing among implicit transactions.

In conclusion, TCC can perform similarly to its predecessor LO, but false sharing seriously degrades performance of most applications. On 32 CPUs, TCC performs an average of 15% worse than LO, though only 11% worse if you ignore `ssca2` with its SERIALIZEDCOMMIT.

Clearly, just as parallel commit is useful for LO, it is even more essential in the continuous transaction environment. While more contention was seen for commit permission because of TCC's periodic commit, it did not hinder any application's scalability that was not already affected in LO. This tells us that while more speculative buffering would probably reduce contention by allowing longer implicit transactions, the level provided by our default cache parameters is sufficient.

5.4.3 Coherence Granularity: Line-level versus Word-level

To combat the false sharing seen in TCC-BASE, we evaluated TCC-WORD and compared their performance in Figure 5.2. In general, word granularity positively affects those applications with false sharing between transactions, but negatively affects those applications heavily dependent on cache-to-cache transfers. Detailed results follow.

bayes

Reducing the false sharing in `bayes` seems to significantly improve its performance, as fewer violations have a favorable effect on transaction scheduling. `bayes` represents the most dramatic improvement, even to surpassing LO's performance (see Table 5.2), but this may be due to fortunate scheduling more than architectural improvements.

genome

In `genome`, we see the effect of decreased Violate and Violate Stall time due to the elimination of false sharing. However, we also see a slight increase in Memory time due to the reduction in cache-to-cache transfers.

intruder

`intruder` does not have a lot of false sharing (as evidenced by TCC-BASE's similar performance to LO) and so the principle effects we see in TCC-WORD is the reduction in cache-to-cache transfers (slight increase in Memory time and the larger increase in Violate Stall).

kmeans

`kmeans` is a unique case: because TCC-WORD avoids the little false sharing there is in `kmeans`, there are slightly fewer violations, meaning more transactions attempting to commit at once. This causes more contention for commit permission (Validate time) as CPU counts increase.

labyrinth

`labyrinth`'s little false sharing prevents TCC-WORD from demonstrating its advantages. In fact, its performance gets worse on TCC-WORD because, as we explained in Section 3.3, `labyrinth` has very long transactions and a high contention rate, making

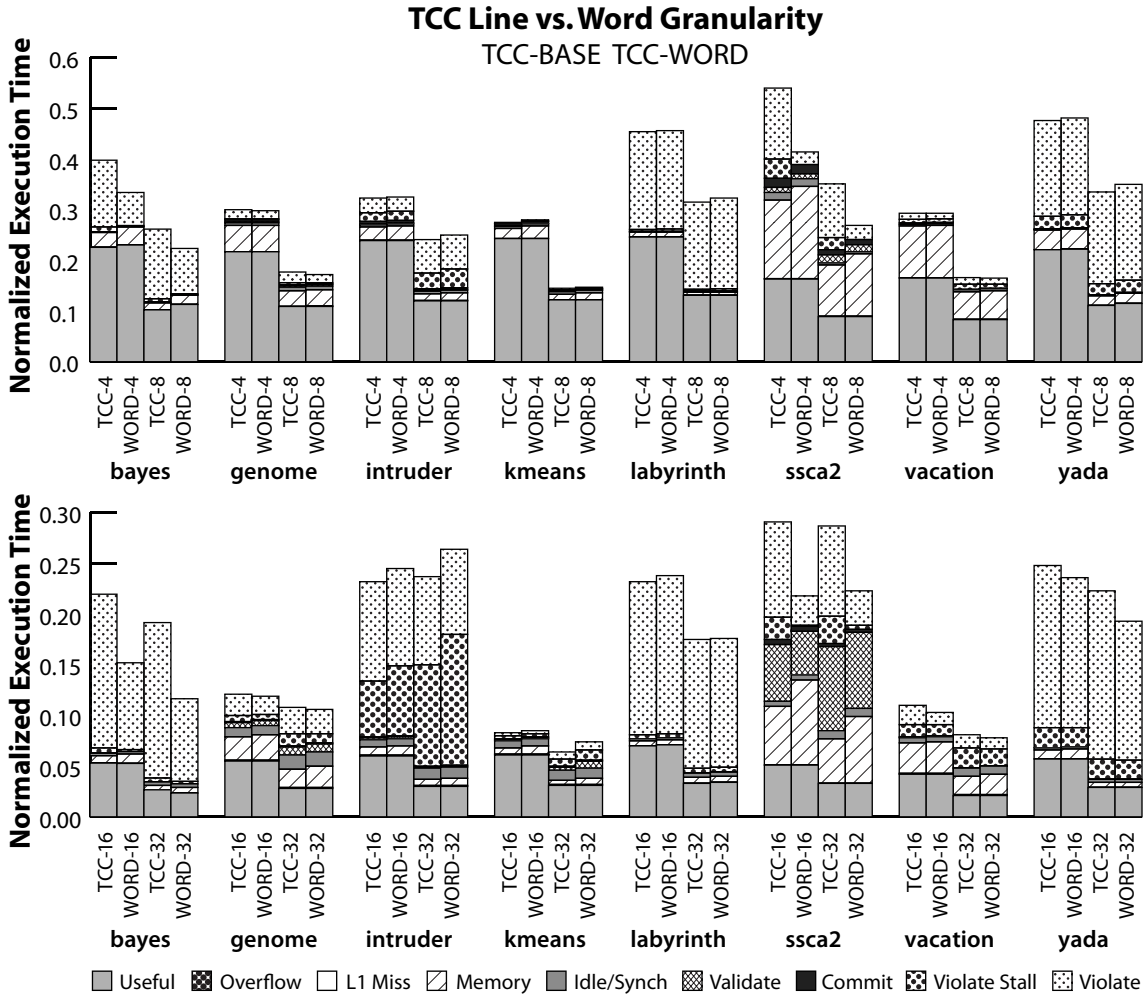

Figure 5.2: Execution time breakdown of STAMP applications on 4–32 CPUs on TCC-WORD compared with TCC-BASE. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

later detection of conflicts waste more work. We see this reflected in the increased Violate time. It could be that better contention management (such as backoff on abort) would improve performance.

ssca2

In the previous section, we noted that `ssca2` on TCC-BASE experienced significant false sharing so we expect TCC-WORD to greatly improve performance, and it clearly does. Unfortunately, the decrease in cache-to-cache transfers (increase in Memory time) prevents performance from reaching that obtained by LO.

vacation

False sharing is eliminated with TCC-WORD, reducing the Violate and Violate Stall cycles, improving performance. However, the difference between TCC-BASE and TCC-WORD is small at all CPU counts.

yada

On TCC-WORD, `yada` experiences significant improvement compared to TCC-BASE. Since the performance improves to be even better than LO's (see Table 5.2) and `yada` spends most of its time in explicit transactions, we can conclude that the explicit transactions experience significant false sharing and TCC-WORD helps avoid it.

In conclusion, word-level granularity significantly improves performance (an average of 27% on 32 CPUs) over line-level granularity. Some applications' performance was hindered by poor contention management, which is further support for our conclusions from Chapter 3, namely that contention management is a chief concern.

5.4.4 Coherence Protocol: Update versus Invalidate

To test the performance of the two coherence protocols, we compared TCC-WORD to TCC-UPDATE and the results are presented in Figure 5.3. Most applications experience little or no change using the update protocol instead of invalidate. Detailed results follow.

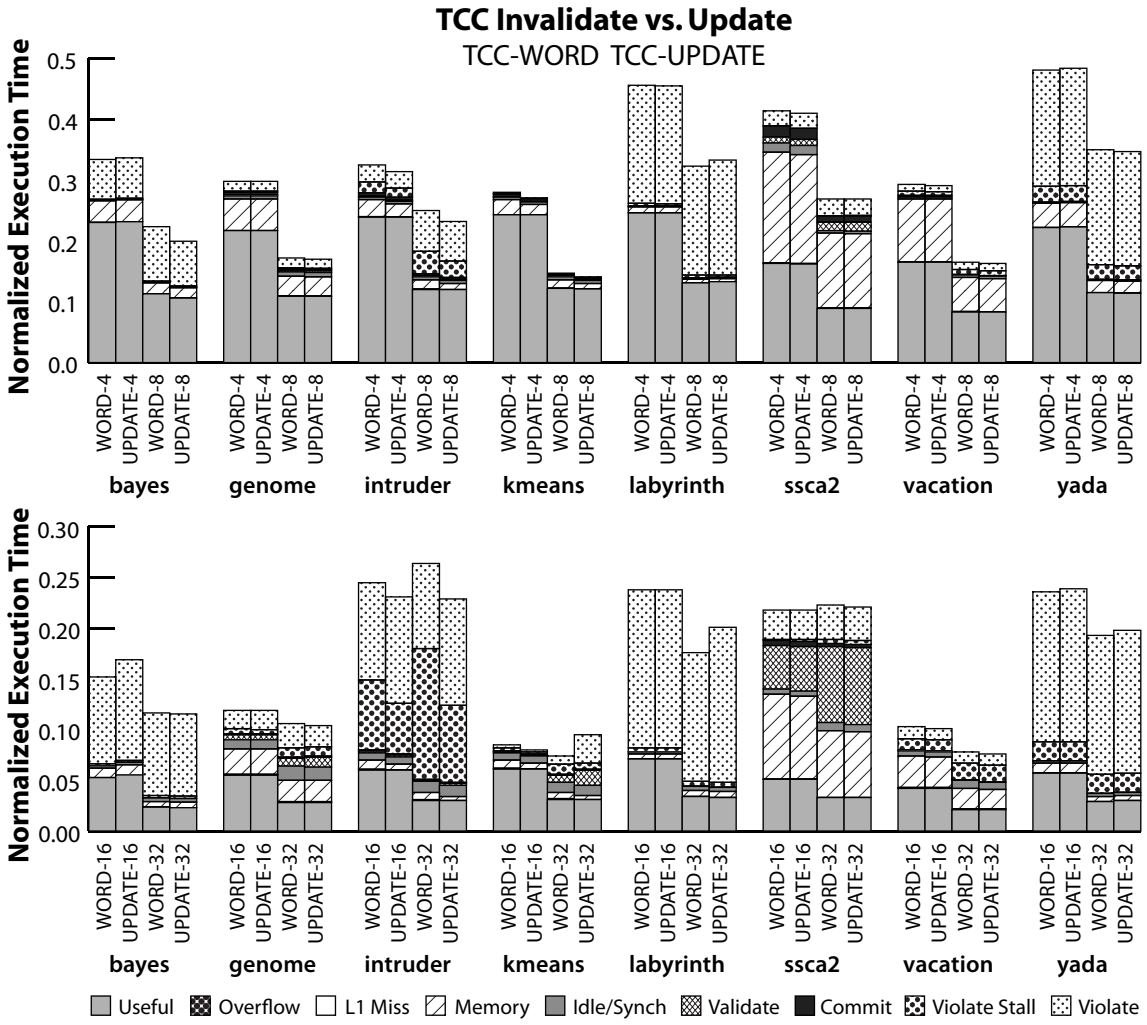

Figure 5.3: Execution time breakdown of STAMP applications on 4–32 CPUs on TCC-UPDATE compared with TCC-WORD. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

bayes

The update protocol makes little impact on the performance of `bayes`, except changing its transaction scheduling. Less time waiting for what would have been refills in TCC-WORD leads to shorter transactions, which alters commit order and therefore convergence rates. Performance is about the same as that of TCC-WORD.

genome

`genome` seems minimally affected by the change in coherence protocol, with a negligible increase in performance over TCC-WORD.

intruder

On `intruder`, we see a significant improvement with TCC-UPDATE as additional load misses are being avoided due to the update protocol. We see this reflected in the decreased Memory and Violate Stall time. Since `intruder` accesses the same data structure with its transactions even after conflicts, this behavior is expected.

kmeans

TCC-UPDATE negligibly improves `kmeans` up to 16 CPUs, reducing Memory and Violate Stall time. However, we see the same pathological behavior at 32 CPUs as we did with TCC-WORD: faster transactions makes for more contention for commit permission (increased Validate time) and possible contention pathologies (increased Violate time).

labyrinth

Surprisingly, more conflicts are present in `labyrinth` using TCC-UPDATE than with TCC-WORD. In fact, as TCC designs get "better" (from TCC-BASE to TCC-WORD to TCC-UPDATE), `labyrinth`'s performance gets worse. Search is the main component of this benchmark and as the long searching transactions become faster (by avoiding refills caused by the invalidate protocol), they contend more with each other due to their immediate restart CM policy, resulting in more Violate time. As in the TCC-WORD case, perhaps backoff on abort would mitigate this effect.

ssca2

Little effect is seen on `ssca2` due to its low contention rate—few conflicts means fewer opportunities for refills caused by the invalidate protocol.

vacation

Like `genome`, `vacation` is not greatly affected by an update protocol, demonstrating a slight improvement in Violate Stall due to avoiding refills.

yada

`yada`'s story is similar to `labyrinth`'s: long transactions and high contention make shorter transactions more contentious. Perhaps backoff would mitigate this effect.

On average, TCC-UPDATE performed 2% worse than TCC-WORD on 32 CPUs. If `kmeans` and `labyrinth`, the two applications with contention management problems, are factored out, TCC-UPDATE performed only 3% better than TCC-WORD. In conclusion, from our experiments, an update protocol is not beneficial. This agrees with our earlier findings in McDonald et al. [66].

5.4.5 Commit Protocol: Commit-through versus Commit-back

Figure 5.4 presents execution time breakdowns of the STAMP applications on TCC-BASE and TCC-BACK, comparing commit protocols. For most applications, the choice of commit-through versus commit-back makes little impact on performance. Detailed results follow.

bayes

In TCC-BACK, we see a reduction in Memory time due to the reduced pressure on the request bus; but more significantly, we see a drop in Violate cycles. Like other `bayes` experiments, changing the memory access patterns changes the transaction schedule. In this case, despite similar abort counts, the shorter transactions in TCC-BACK cause conflicts to be detected earlier, leading to aborted transactions being 50% shorter in TCC-BACK than in TCC.

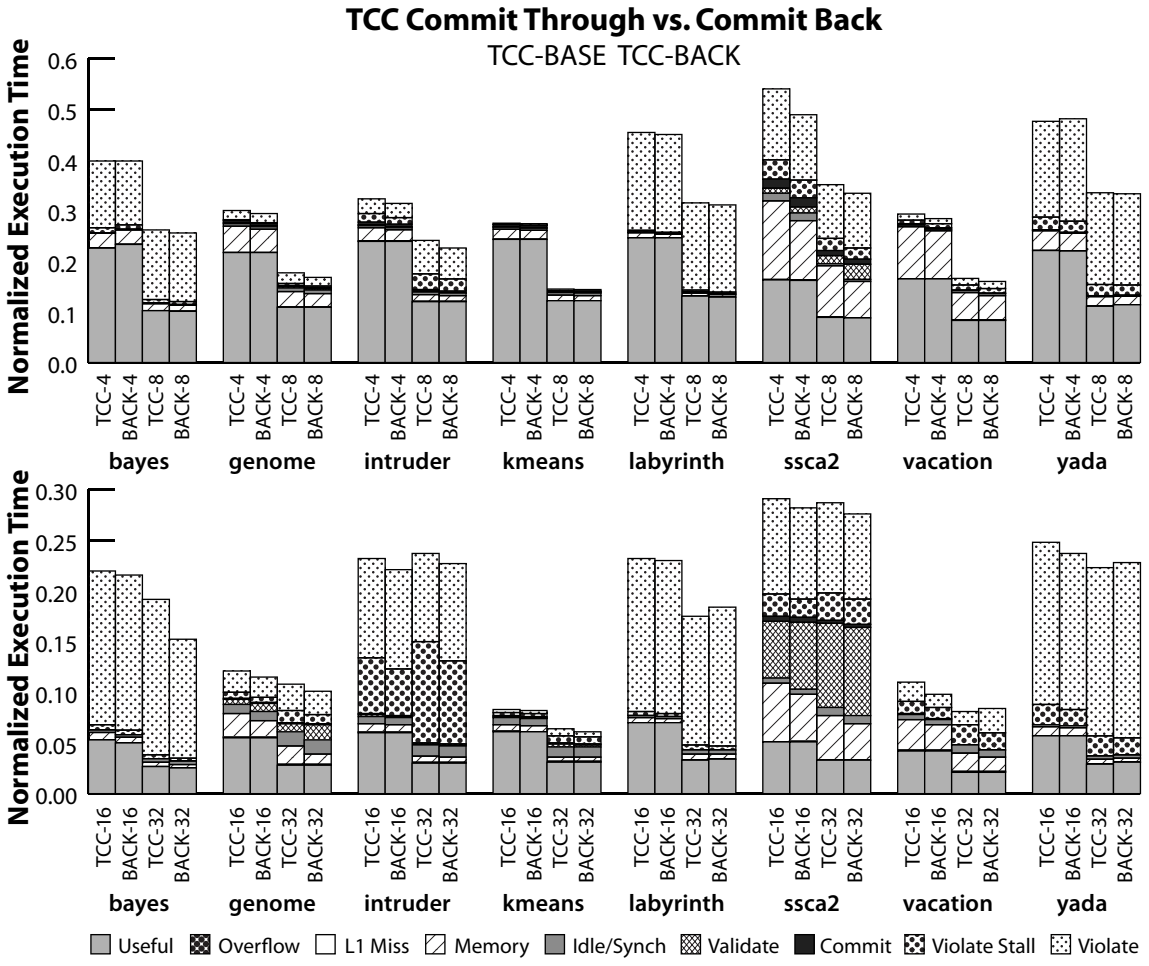

Figure 5.4: Execution time breakdown of STAMP applications on 4–32 CPUs on TCC-BACK compared with TCC-BASE. Normalized to sequential execution. Default simulator parameters (see Table 3.1).

genome

In `genome`, TCC-BACK reduces the time needed for refills because it reduces pressure on the shared cache—the shared cache no longer must store commit data, providing more bandwidth for refills (commit data takes an pipelined 3 cycles to cross the line-sized bus). Performance gains are somewhat tempered by the fact that shorter transactions lead to more contention for commit permission (increased Validate time), just as we saw with `kmeans` on TCC-WORD and TCC-UPDATE.

intruder

Similar to `genome`, `intruder` experiences performance gains in TCC-BACK due to the reduction in memory pressure. Because of `intruder`'s high contention rate, most of the memory accesses are in transactions that later abort, so TCC-BACK's effect is seen predominantly in the reduction of Violate Stall time.

kmeans

Having small transactions and a low contention rate, `kmeans` has little memory pressure to begin with, so TCC-BACK makes little performance impact.

labyrinth

`labyrinth`'s explicit transactions are very long and make few writes per transactional instruction, so performance is not significantly affected by the decrease in memory pressure offered by TCC-BACK. Similar to the effect at 32 CPUs in TCC-UPDATE, TCC-BACK makes `labyrinth`'s transactions faster, increasing contention and resulting in more Violate time. Again, proper contention management may mitigate this.

ssca2

Like in other applications, `ssca2` experiences a decrease in memory pressure, seen in lower Memory and Violate Stall times. This results in a net performance gain, even though faster transactions do increase contention for commit permission (increased Validate time).

vacation

Again, we see the same patterns in `vacation`: decreased Memory and Violate Stall components due to avoiding the storage of commit data in the shared cache.

yada

Like `yada` under TCC-UPDATE, little improvement is found with TCC-BACK since very little of `yada`'s execution time is spent accessing memory. Also like TCC-UPDATE, TCC-BACK does reduce the execution time of its long transactions enough to create increased contention at high CPU counts.

In conclusion, commit-back does not perform significantly better than commit-through (only an average of 3% on 32 CPUs) and since it is difficult to combine with word granularity, which did improve performance, it may not be worth implementing. However, commit-back may offer an energy advantage since data is sent only when and where it is needed. Additionally, in large scale systems, a broadcast interconnect may not be feasible, forcing the need for commit-back. It is worth noting that Commit time was not affected by this experiment because our bus is one cache line wide. On thinner buses, where each commit-through packet would take longer to process, we expect TCC-BACK to improve more compared to TCC-BASE. Finally, just as we saw with the previous experiments, TCC-BACK suffers from SERIALIZEDCOMMIT and would benefit from parallel commit.

5.4.6 Bus Utilization

Because TCC is always executing transactions, and those transactions are frequently committing, we expect it to tax interconnect resources more than LO. Figure 5.5 presents the bus utilization breakdowns on 32 CPUs between TCC-BASE and LO-BASE.

Clearly, neither the LO nor TCC systems push the bus to its limits, and the TCC systems utilize the bus only slightly more (an average of 5% more). `ssca2` shows a decrease in bus utilization because the TCC system had many transactions waiting to validate, increasing the time the bus was idle.

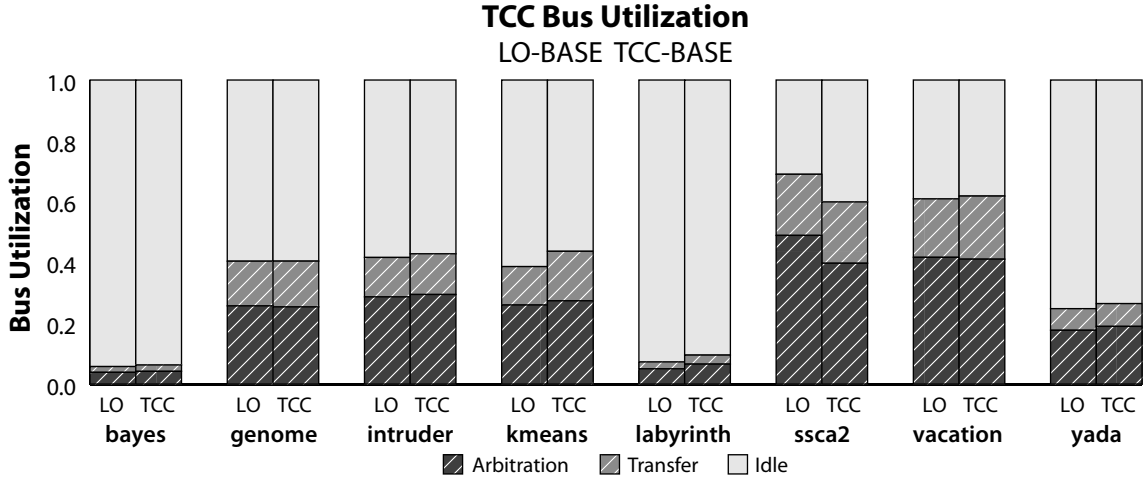

Figure 5.5: Comparing TCC-BASE's and LO-BASE's bus utilization breakdowns on 32 CPUs. Default simulator parameters (see Table 3.1).

In conclusion, even though TCC's continuous transactions do use more interconnect resources than a traditional TM system, its requirements are well within the capabilities of a modern multicore system. These findings echo our results in McDonald et al. [66], where we examined TCC's bus utilization compared to a traditional MESI system using SPEC, SPLASH, and SPLASH-2 applications.

5.5 Conclusions

We investigated the implementation and performance of the first continuously transactional system, called TCC, as compared to traditional TM methods. Using the STAMP benchmarks, we found that TCC's performance, while generally not as good as LO's, is adequate for transactional execution. Furthermore, using the same level of speculative buffering and bus bandwidth as available in LO systems is adequate for TCC. Finally, as we saw with LO, TCC and its design alternatives would benefit from parallel commit to address the SERIALIZEDCOMMIT pathology.

We also studied implementation alternatives for TCC. We found that tracking state at the word granularity is best, despite some scheduling effects that may be mitigated by

proper contention management. We showed that the update commit protocol was better on most applications, but performance improvements may not justify the additional complexity. Finally, we examined a commit-back protocol and found it useful, but not as useful as word granularity, and the hardware costs most likely preclude combining with word granularity. In conclusion, TCC with word-based granularity, an invalidate coherence protocol, and commit-through is probably the best design. This agrees with our findings in McDonald et al. [66].

Overall, TCC provides adequate parallel performance for transactional applications in addition to the simpler parallel programming model. Continuous transactions continues to be explored actively in both the hardware and software communities. UIUC's Bulk Multicore Architecture [100] and Microsoft Research's Automatic Mutual Exclusion [54] projects have recognized the potential advantages and sought to develop practical continuous transaction systems.

TCC Speedups Across Design Alternatives

	4 CPUs					8 CPUs				
	LO	BASE	WORD	UPDATE	BACK	LO	BASE	WORD	UPDATE	BACK
bayes	2.45	2.51	2.99	2.97	2.51	3.76	3.82	4.47	5.01	3.91
genome	3.43	3.31	3.33	3.34	3.37	6.00	5.63	5.76	5.83	5.90
intruder	3.19	3.08	3.06	3.18	3.17	4.44	4.16	4.00	4.30	4.41
kmeans	3.64	3.63	3.57	3.67	3.64	6.94	6.89	6.75	7.00	6.94
labyrinth	2.22	2.19	2.19	2.19	2.21	3.20	3.18	3.10	3.01	3.22
ssca2	2.79	1.84	2.39	2.41	2.02	4.67	2.85	3.72	3.74	2.99
vacation	3.50	3.40	3.40	3.42	3.51	6.34	6.03	6.07	6.11	6.24
yada	2.05	2.10	2.07	2.07	2.08	2.99	2.98	2.86	2.88	3.01
	16 CPUs					**32 CPUs**				
	LO	BASE	WORD	UPDATE	BACK	LO	BASE	WORD	UPDATE	BACK
bayes	4.65	4.55	6.58	5.97	4.64	6.06	5.22	8.58	8.66	6.56
genome	9.27	8.24	8.35	8.41	8.66	12.08	9.30	9.44	9.68	9.77
intruder	4.56	4.32	4.08	4.31	4.51	4.48	4.22	3.78	4.36	4.42
kmeans	12.54	12.13	11.84	12.30	12.25	18.85	15.84	13.59	10.49	16.70
labyrinth	4.23	4.28	4.18	4.18	4.32	5.82	5.73	5.69	4.98	5.41
ssca2	5.92	3.43	4.55	4.57	3.55	5.89	3.50	4.50	4.52	3.63
vacation	10.61	9.10	9.74	9.83	10.24	14.13	12.44	12.79	13.04	11.72
yada	4.08	4.03	4.23	4.17	4.22	4.81	4.46	5.17	5.03	4.38

Table 5.2: Speedups for each application on each of the TCC systems and LO, from 2–32 CPUs. Simulation parameters are the defaults found in Table 3.1.

Chapter 6

Conclusions and Future Work

In this thesis, I set out to establish that Transactional Memory is an attractive alternative to lock-based synchronization, that Lazy-Optimistic should be the preferred TM design, that rich semantics for TM are needed to implement modern OS and language features, and that hardware support for continuous transactions is not only useful but practical.

In Chapter 3, I compared transactional memory designs and concluded that Lazy-Optimistic was consistently the best-performing and lowest-complexity TM design. Furthermore, I found that TM's performance was comparable to that of conventional synchronization techniques. Because of the work of myself and others, TM has begun to emerge in commercial processors like Azul's Java Compute Appliances [29] and Sun's Rock [33]. Both of those designs implement LO systems. Future directions should include exploring low-cost virtualization mechanisms in such HTM systems and of course, using these production systems to evaluate TM programming languages.

In Chapter 4, I argued that four mechanisms are needed to implement real-world transactional memory: two-phase commit, software handlers, closed- and open-nesting, and non-transactional loads and stores. I showed how they could be implemented with low overhead to support modern operating systems and languages. Future work would be to use these mechanisms to implement a complete TM system and evaluate it.

Finally, in Chapter 5, I described and examined the first continuous transaction architecture, Transactional Coherence and Consistency (TCC), which is based on LO. I

found that its performance, while lower than LO, can be improved by choosing word-level coherence granularity, and is adequate for transactional execution. Microsoft Research's AME and UIUC's Bulk projects continue to explore the hardware and software advantages and implementations of continuous transactions.

The future of transactional memory is a bright one, with many researchers in both academia and industry actively pursuing a number of projects. More work is needed to explore how TM applications will be built by programmers who exclusively use transactions, and results from that work may influence HTM design philosophy. Also, as TM becomes a reality in commodity systems, educators should examine their programming curricula so as to adequately prepare the next generation of coders to exploit parallelism via transactions.

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] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174, New York, NY, USA, 2008. ACM.

[4] K. Agrawal, C. E. Leiserson, and J. Sukha. Memory models for open-nested transactions. In *MSPC: Workshop on Memory Systems Performance and Correctness*, October 2006.

[5] A. Ahmed, P. Conway, B. Hughes, and F. Weber. Amd hammer. In *Conference Record of Hot Chips 14*, Stanford, CA, August 2002.

[6] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, 2005.

[7] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen,

T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[8] 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.

[9] 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.

[10] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, June 2000.

[11] 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.

[12] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. June 2007.

[13] 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.

[14] 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.

[15] B. D. Carlstrom. Embedding Scheme in Java. Master's thesis, Massachusetts Institute of Technology, February 2001.

[16] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional Collection Classes. In *Proceeding of the Symposium on Principles and Practice of Parallel Programming*, March 2007.

[17] 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.

[18] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 278–289, New York, NY, USA, 2007. ACM Press.

[19] 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.

[20] H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. TAPE: a transactional application profiling environment. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 199–208, New York, NY, USA, 2005. ACM.

[21] 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.

[22] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster

computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[23] 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.

[24] J. Chung. *System Challenges and Opportunities for Transactional Memory*. PhD thesis, Stanford University, Jun 2008.

[25] J. Chung, W. Baek, N. G. Bronson, J. Seo, C. Kozyrakis, and K. Olukotun. Ased:availability, security, and debugging support using transactional memory. In *20th ACM Symposium on Parallelism in Algorithms and Architectures*. Jun 2008.

[26] 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.

[27] 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.

[28] J. Chung, J. Seo, W. Baek, C. Cao Minh, A. McDonald, C. Kozyrakis, and K. Olukotun. Improving software concurrency with hardwrae-assisted memory snapshot. In *SPAA '08: Proceedings of the Twentieth Annual ACM Symposium on Parallel Algorithms and Architectures*, 2008.

[29] C. Click. A tour inside the Azul 384-way Java appliance. Tutorial held in conjunction with the Fourteenth International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2005.

[30] Cray. *Chapel Specification*. February 2005.

[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] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[33] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[34] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[35] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. University of Rochester, October 2005.

[36] 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.

[37] L. Hammond. *Hydra: A Chip Multiprocessor with Support for Speculative Thread-Level Parallelization*. PhD thesis, Stanford University, 2002.

[38] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March/April 2000.

[39] 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.

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

[41] 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.

[42] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, July 2005. ACM Press.

[43] 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.

[44] T. Harris and S. Stipic. Abstract nested transactions. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[45] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[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, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[48] 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.

[49] B. Hertzberg. *Runtime Automatic Speculative Parallelization of Sequential Programs*. PhD thesis, Stanford University, 2009.

[50] M. D. Hill, D. Hower, K. E. Moore, M. M. Swift, H. Volos, and D. A. Wood. A case for deconstructing hardware transactional memory systems. Technical Report CS-TR-2007-1594, University of Wisconsin-Madison, Department of Computer Sciences, June 2007.

[51] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[52] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA '08: Proceedings of the 35th annual international symposium on Computer architecture*, New York, NY, USA, 2008. ACM Press.

[53] J. Huh, J. Chang, D. Burger., and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[54] M. Isard and A. Birrell. Automatic Mutual Exclusion. In *11th Workshop on Hot Topics in Operating Systems*, May 2007.

[55] Java Grande Forum, *Java Grande Benchmark Suite*. `http://www.epcc.ed.ac.uk/javagrande/`, 2000.

[56] *JBus Architecture Overview*. Technical report, Sun Microsystems, April 2003.

[57] T. Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA, August 1986. ACM Press.

[58] P. Kongetira. A 32-way multithreaded Sparc processor. In *Conference Record of Hot Chips 16*, Stanford, CA, August 2004.

[59] 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.

[60] J. Larus. It's the software stupid. Talk at the Workshop on Transactional Systems, April 2005.

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

[62] V. Luchangco and V. Marathe. Transaction Synchronizers. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. University of Rochester, October 2005.

[63] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.

[64] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 18–29, New York, NY, USA, October 2002. ACM Press.

[65] 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.

[66] 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.

[67] W. mei W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th International Symposium on Computer Architecture*, June 1987.

[68] P. Montesinos, L. Ceze, P. Montesinos, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA '08: Proceedings of the 35th annual international symposium on Computer architecture*, New York, NY, USA, 2008. ACM Press.

[69] 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.

[70] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, New York, NY, USA, 2006. ACM Press.

[71] J. E. B. Moss. Nesting transactions: Why and what do we need? In *TRANSACT Invited Talk: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[72] J. E. B. Moss. Open Nested Transactions: Semantics and Support. In *Poster at the 4th Workshop on Memory Performance Issues (WMPI-2006)*. February 2006.

[73] J. E. B. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. University of Rochester, October 2005.

[74] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. Sigrace: Signature-based data race detection. In *ISCA '09: Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[75] V. Nagarajan and R. Gupta. Ecmon: Exposing cache events for monitoring. In *ISCA '09: Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[76] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 68–78, New York, NY, USA, 2007. ACM Press.

[77] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 184–196, October 2002.

[78] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. pages 111–122, 2002.

[79] 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.

[80] 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.

[81] H. Ramadan, C. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 246–257, Nov. 2008.

[82] 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.

[83] M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM Press.

[84] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, 1995.

[85] 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.

[86] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

[87] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July 2004.

[88] 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.

[89] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, October 2008. ACM.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*.

[95] R. Singhal. Inside intel core microarchitecture (nehalem). In *Conference Record of Hot Chips 20*, August 2008.

[96] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*. June 2006.

[97] Standard Performance Evaluation Corporation, *SPEC CPU Benchmarks*. `http://www.specbench.org/`, 1995–2000.

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

[99] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-level Speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, British Columbia, Canada, June 2000.

[100] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesino, W. Ahn, and M. Prvulovic. The bulk multicore architecture for improved programmability. *Communications of the ACM*, 2009.

[101] S. Wee. *ATLAS: Software Development Environment for Hardware Transactional Memory*. PhD thesis, Stanford University, 2008.

[102] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In M. Odersky, editor, *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, June 2004.

[103] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *SPAA '08: Proceedings of the twentieth annual ACM symposium on Parallel algorithms and architectures*, 2008.

[104] 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.

[105] 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*, February 2007.