LOG-BASED TRANSACTIONAL MEMORY

by

Kevin E. Moore

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

2007

Abstract

A pressing challenge in the movement to an explicitly parallel model of computing is to give programmers, compiler writers and language designers a more powerful means of synchronization. My doctoral research attacks this problem by investigating practical, high performance transactional memory systems. Transactional memory gives programmers the ability to declare the synchronization properties their programs need, without requiring that they develop a mechanism (e.g., a locking scheme) to enforce these properties. Furthermore, transactions may be nested to allow programmers to build thread-safe libraries without exposing implementation details such as locking conventions to higher levels of software.

Transactional memory systems may be implemented in software, in hardware, or in a combination of the two. Typically, transactional memory systems do not limit the amount of memory a transaction may access nor the length of time a transaction may run. Though intuitive for programmers, that model is poorly suited for direct implementation in hardware, which is limited to structures of fixed size. Ideally, an implementation of transactional memory should: (1) make the expected common case (short transactions that commit) fast; (2) defer rare and difficult-to-implement cases to software; (3) allow transactions of any size and duration; and, (4) allow programmers to nest transactions to arbitrary depths.

In my doctoral research, I developed Log-Based Transactional Memory (LogTM), a transactional memory system that combines software-based *version managemen*t (with limited hardware support) and conservative hardware *conflict detection* to support arbitrary-size transactions with limited hardware. Version management requires maintaining multiple versions of data values: old values, the values at the start of the transaction, and new values, which are generated inside a transaction. Any memory location modified in a transaction will hold its new value if the transaction commits, and its old value if the transaction aborts. LogTM performs version management by eagerly updating memory in place during transactions and sav-

ing old values in a per-thread transaction log. No further action is needed to commit a transaction since new values are kept in place. To abort a transaction, LogTM restores saved values from the log. Fortunately, (for most workloads) aborts are rare. Because aborts are infrequent, LogTM can reduce hardware complexity by performing aborts in software without degrading performance. Conflict detection identifies overlaps between the write set of each transaction with the read set or write set of other concurrent transactions. Like other hardware transactional memory systems, LogTM detects conflicts on cached data by augmenting the cache coherence mechanism, e.g., by adding a read (R) and write (W) bit to each cache line. LogTM implementations extend this mechanism (e.g., with sticky states, which enable the directory to continue to send coherence messages to a processor for a blocks it has evicted from its cache) so that processors can conservatively detect conflicts even for blocks not present in the cache.

Before I began work on LogTM, the only published hardware transactional memory systems relied heavily on the cache. The size and associativity of the cache limited the scope of transactions. LogTM's unique log-based version management combined with innovative sticky states allow it to break this dependence on the cache without adding complex hardware. Unlike previous schemes, which relied on keeping old values in memory and storing new values in the cache, LogTM stores new and old versions in separate memory locations, which may be cached and evicted independently. Like the stack, the transaction log is a part of a thread s virtual memory and is effectively unbounded. Sticky states, although part of the coherence mechanism, break the dependence between conflict detection and caching. By allowing conflict detection on blocks after eviction, sticky states enable transactions whose read and write sets exceed the capacity or associativity of the cache. Perhaps most importantly, LogTM guarantees that transactions appear atomic to applications, but it allows some lower-level software to observe transactions' intermediate values. This allows LogTM to involve software in maintaining the atomicity of transactions. LogTM uses hardware to perform the most performance-critical tasks, tracking read and write sets and detecting conflicts, but leaves rare and complicated tasks, such as aborting transactions, to software.

Acknowledgements

Before I begin, let me say a word of thanks to my family friends, colleagues and sponsors without whose support, love and guidance I would never have completed this dissertation nor even began my studies in the first place.

First, to everyone who worked with me developing, extending and evaluating LogTM, Jayaram Bobba, Mark Hill, Ben Liblit, Michael Marty, Michelle Moravan, Mike Swift, Haris Volos, David Wood and Luke Yen, it has been my great pleasure to work with all of you. Your insight and hard work has expanded this project far beyond anything I expected when I first discussed the idea with David. Alaa Alameldeen, Brad Beckman, Milo Martin and Min Xu, thank you for being excellent colleagues and friends and for creating the simulation infrastructure on which I relied for all of my experiments.

Martin Karlsson, Eric Hagersten and Michael Koster, thank you for your hard work and insights during our collaboration characterizing Java workloads. The experience I gained from that project guided all my subsequent research. I also sincerely thank the National Science Foundation for sponsoring my research with grants CCF-0085949, CCR-0105721, EIA/CNS-020528, and CCR-03248 and Sun Microsystems and IBM for their support of my research and that of my colleagues in the Multifacet group at the University of Wisconsin.

But, my doctoral research is merely a small part of my education. So, let me also thank the many professors and teachers who have inspired me to pursue my education over the years. To my professors at the University of Wisconsin, especially: Jim Goodman, Mary Vernon, John Kidwell, Anuj Desai and Pilar Ossorio, thank you for your excellent teaching, guidance and enthusiasm in and out of the classroom. And, to the teachers whose lessons prepared me for graduate school, especially Linda Chittenden, Rita Chaney, Dennis Duncan, Jackie Caplan-Auerbach, Marty Talant, and John Board, thank you for teaching me to think critically, write clearly and to value learning for its own sake.

No two professors have taught me more than Mark Hill and my advisor, David Wood. Mark, I can't imagine that you realize how much you'e taught me (or, perhaps how much I realize that you've taught me). Thank you for teaching me so very much about architecture, research, organizing and expressing ideas and about working effectively with others (especially my advisor). And, David, over the past several years, you have been my harshest and most helpful critic. Thank you for guiding me and motivating me. But most of all, thanks for being a role model—for showing me that even those who are not naturally organized can be successful.

Let me also thank the many friends who made my time in Madison so special. Amy, Laurie, Andrew, Ian, Irene, Karen, Drew (and you too Drew), thanks for the "Tuesday" dinners, market Saturdays and concerts on the square. To Mark, Andrew, Nick, Brad and Ina, thanks for getting me (and keeping me) running and biking, which saved my sanity many times. And thanks to all of you, for the constant support, advice and friendship that made my time in Madison wonderful. But, there is one friend to whom I owe the most thanks for my time in Madison. Dan, without your encouragement and advice I would have never come to Wisconsin or continued to study architecture. Thank you so very much for introducing me to Wisconsin and for being such a valuable friend and mentor for many years.

Of all the influences in my life, nothing has done more to shape me, my interests and my character than my family. Allison, thank you for always cheering me up when I was down. Mom, thank you so much for sharing with me your love of learning and for teaching me nearly all that I know about writing and language. Dad, you are my role model for living. Thank you for teaching me not to lose sight of what matters most, and for introducing me to computers. And, thank you all for encouraging my curiosity and creativity my whole life. I am who I am because of you, your support and your love.

Finally, Jenny, for all your support, comfort, humor, critiquing, proofreading, and most of all for your love these many years, I cannot thank you enough. This dissertation is as much yours as it is mine; you have shared all my successes and struggles throughout the course of writing it.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel programming is difficult. Transactional Memory can make it less so. But, supporting transactional memory in hardware is too expensive and supporting it in software is too slow. As a result, transactional memory systems have evolved in two ways: (1) hardware systems, to which software is added to "virtualize" hardware's physical limits and (2) software systems to which some hardware is added to improve speed. Instead, LogTM takes a holistic approach—it implements unbounded transactional memory using software (with some hardware assistance) to store and recover multiple versions of data (which may require extensive state), and hardware to detect transaction conflicts (which must be performed quickly). By implementing performance-critical aspects of transactional memory in hardware and the rest in software, LogTM requires few hardware changes to implement, yet performs comparably to all-hardware implementations.

1.1 Motivation

The remarkable performance gains that have fueled the computer industry for many years are today threatened by two trends that are pulling hardware and software in opposite directions. The first is an increasing emphasis on software reliability. As our society comes to rely ever more on computer systems, the cost of software failures is increasing. At the same time, the expanding role of computer systems has led to more complex software, making testing and debugging programs correspondingly more difficult. The second trend is the industry-wide shift towards thread-level parallelism. Recently, all high-performance processor makers have turned to chip multiprocessors (CMP), effectively making every computer sold today a multiprocessor. Many have also adopted multithreading, which allows each processor to execute multiple tasks concurrently. Unlike the gains in single-threaded performance, increased thread-level parallelism only speeds up parallel software. Unfortunately, the task of writing reliable software is only made more challenging with the introduction of parallelism. The chip industry's current solution, increasing performance by increasing thread-level parallelism, creates a dilemma for programmers: either ignore parallelism and face stagnant performance, or embrace parallelism and face a new class of bugs.

1.2 Synchronization

In addition to the challenges of writing a correct sequential program, to write a correct parallel program, one must first divide the program's work into separate tasks, so that it may be performed in parallel by a number of *threads* of control. Next, one must ensure that each thread has access to the data on which it operates. Finally, the writer of a parallel program must coordinate these threads in some way to ensure that they access shared data structures in a consistent manner. To implement this coordination, parallel program developers use *synchronization* to impose ordering constraints on the actions of various threads. In practice, determining the synchronization needed in a program, and devising a scheme to enforce it, is often one of the most challenging aspects of parallel programs and a source of costly bugs.

1.2.1 Serialization

One intuitive model for synchronization is serial execution. Accesses from each thread to a shared data structure are performed in *critical sections*. If critical sections execute serially with regard to other critical sections, programmers may perform operations that leave shared data structures in inconsistent states temporarily, without exposing those inconsistencies to other threads. Dijkstra described this problem as the *mutual exclusion* problem [19]; only one thread may execute a critical section at a time.

```
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

(a)

```
void move(T s, T d, Obj key){
  atomic {
   tmp = s.remove(key);
   d.insert(key, tmp);
  }
}
```

(b)

FIGURE 1-1. Composability of Transactions v. Locks: (a) moving an element from one hash table to another using locks, (b) using transactions.

On today's hardware, ensuring mutual exclusion requires considerable effort from software. Programmers must establish and maintain a convention to record entry to and exit from critical sections and to prevent overlap between them. The current state of the art in parallel programming is to enforce mutual exclusion using *locks*, words in memory that, by software convention, represent the ability of threads to access certain data. Typically, a lock is associated with a set of data. To access a shared data structure, a thread must enter a critical section by acquiring the lock that protects that structure.

1.2.2 The Trouble with Locks

When used properly, mutual exclusion via locks provides an efficient mechanism for preventing inconsistent accesses by concurrent threads. Using locks, however, is notoriously difficult, especially in large software systems built out of many separate components. Ideally, programmers should be able to compose large programs out of smaller pieces of code (e.g., reusable libraries) without knowing anything about their internal mechanisms. Locks, however, impede such composition. Consider the example in Figure 1-1, moving an item from one hash table to another. In the lock-based version (Figure 1-1-a), locks are needed to prevent other threads seeing a state where the moving object is not present in either table. While these locks may seem like a reasonable solution, even the simple example in Figure 1-1-a could result in a deadlock. If two threads call `move` concurrently on the same two tables, but with the source and destination reversed, both threads could acquire the lock on the source table, then block while trying to acquire the lock on the destination table.

Additionally, programmers using locks must balance the desire for concurrency with the overhead of acquiring and releasing many locks. Choosing fine-grain locks leads to greater concurrency, but incurs greater overhead and increases the chances of synchronization bugs. Conversely, choosing coarse-grain locks reduces lock overhead, but can reduce concurrency as more threads and processors are allocated to the program.

1.2.3 Transactions

An alternative to the mutual exclusion model of synchronization is transactional programming. As developed in the database community [27], a *transaction* is defined as a transformation of state that is: atomic, consistent, isolated, and durable. Atomicity guarantees that either all of the transaction's actions will complete, or none will. Consistency means that a transaction that finds the system in any consistent state will leave the system in a (possibly different) consistent state and is the burden of the programmer. Isolation means a transaction will execute as if all other threads or processes are stopped while it is running. Finally, durability guarantees that a transaction, once completed, will not be undone, even in the event of a system failure. In transactional programming, all operations that access shared state are part of some transaction.

1.3 Transactional Memory

Transactional Memory [34, 44] extends the transactional programming model to shared-memory programs. In place of critical sections protected by locks, programmers specify atomic transactions. Like a database system, a transactional memory system guarantees that transactions will execute atomically and in isolation. Unlike database systems, however, transactional memory systems do not guarantee that transactions are durable. Forgoing the durability guarantee allows transactional memory systems to implement transactions with much less overhead than database systems.

Compared to critical sections, transactions have several advantages. First, transactions free programmers from reasoning about the correctness and performance of their locking scheme. Second, in addition to isolation, which proper mutual exclusion ensures, transactions provide atomicity, assuring the programmer that, even in the event of a failure, shared data structures will not be left in an inconsistent state. The main advantage of transactional memory, however, is that—unlike lock-protected critical sections—transactions compose naturally. In contrast to the lock-based move method (Figure 1-1-a), the transactional move method (Figure 1-1-b), composes naturally with any internal transactions in the table class. The underlying transaction system will automatically serialize calls to `move` that operate on the same table without risking deadlock.

1.3.1 Limitations of Software Transactional Memory

Transactional memory can be implemented in hardware, software, or a combination of the two. Software Transactional Memory (STM) systems [30, 32, 69, 72], require no changes to existing hardware and are therefore the easiest to implement. Most STMs, however, suffer from two serious drawbacks, (1) poor performance and (2) weak atomicity. The fastest STM systems released to date are at best comparable to and often slower than the best lock-based algorithms for many applications [1]. More importantly, most STMs support only *weak atomicity* [8], meaning that transactions are isolated only from memory references that are part of other transactions. Weak atomicity can result in nonintuitive behavior and expose details of the STM implementation [44, 73]. As a result, supporting only weak atomicity will likely make writing and debugging more challenging, eroding the primary benefit of transactional memory. Thus far, the only high-performance STM to support strong atomicity incurs a 40% overhead over unsynchronized code [73].

Although many software developers may be willing to sacrifice performance for correctness (e.g., preventing deadlocks) the motivation for making a program run in parallel is to improve its performance.

1.4 Hardware Transactional Memory

Hardware Transactional Memory (HTM) has the potential to provide both high performance and strong atomicity. HTM systems typically leverage cache coherence mechanisms to provide transactional isolation much more efficiently than STMs. HTM systems are not only more efficient than STMs, but are more efficient than lock-based synchronization for most applications. HTMs eliminate the overhead of acquiring and releasing fine-grained locks. Because HTMs typically leverage the cache coherence mechanism, they naturally check all memory references against any active transactions. Thus, they provide strong atomicity with little or no additional overhead.

1.4.1 Implementing Hardware Transactional Memory

Implementing HTM primarily requires *version management* and *conflict detection*, both of which are often implemented by augmenting processor caches. *Version management* handles the simultaneous storage of both *new* data (to be visible if the transaction *commits*) and *old* data (retained if the transaction *aborts*). A common approach for version management is to use the cache to store new values, while old values remain in memory. *Conflict detection* signals an overlap between the *write set* (data written) of one transaction and the write or *read sets* (data written or read) of other concurrent transactions. Most HTMs leverage the cache coherence mechanism to perform conflict detection.

HTMs that use caches for version management and conflict detection have been shown to excel for transactions that execute for short durations and touch only small amounts of memory [29, 34, 65]. Long-running transactions that touch larger amounts of memory, however, present a challenge to such systems because both version management and conflict detection break down when data involved in a transaction leave the cache. Supporting transactions of arbitrary size and run time in hardware is difficult because hardware is constrained by the physical size of its structures. Implementing unbounded transactions with bounded physical structures requires *virtualizing* conflict detection and version management.

1.4.2 Virtualizing HTM

Virtualization is the use of system software to provide the illusion of unbounded or idealized hardware to application programs. When physical resources are exhausted, software compensates, allowing applications to continue to function albeit more slowly. For example, virtual memory gives application programs the illusion of a large private memory space even when physical memory is limited and shared by many processes. It allows programmers to reason about common case memory behavior knowing that their programs may exceed the physical resources of the system so long as they do so rarely.

For transactional memory, virtualization requires abstracting the limitations of hardware structures used to perform version management and conflict detection and ensuring that transactions do not break existing virtualization mechanisms such as virtual memory and time slicing.

1.5 Problem Statement

Transactional memory is an emerging programming technique that promises to ease multi-threaded programming provided that transactional memory systems can provide sufficient performance without sacrificing the convenience of the transactional model. Currently, it appears that hardware support will be necessary to implement transactional memory systems that support strong atomicity and still execute parallel programs as efficiently as lock-based synchronization. All-hardware implementations of transactional memory are not practical; they either restrict the execution model by limiting the size or duration of transactions, or introduce unnecessary expense by allocating hardware resources that are used infrequently. Therefore, an implementation of transactional memory should strive to:

- make the common case fast, namely short transactions that commit;
- defer rare and difficult-to-implement cases to software; and
- allow—through a combination of hardware and software—transactions of any size and duration.

1.6 LogTM

To this end, I propose *Log-based Transactional Memory (LogTM)*, a transactional memory system that combines software-based version management (with limited hardware support) and conservative hardware conflict detection to support arbitrary-sized transactions with limited hardware. LogTM performs version management by creating a per-thread *transaction log* in cacheable virtual memory, which holds the virtual addresses and old values of all memory blocks modified during a transaction. In LogTM, a transaction commits by discarding the log (resetting a log pointer) and flash clearing some local coherence state. No other work is needed, because new values are already in place. A transaction aborts by walking the log in software to restore values.

Contributions: In developing LogTM, I make the following contributions:

- I develop and evaluate a transactional memory system that supports unbounded transactions with limited hardware resources.
- I develop a transactional memory system that optimizes for commit, by using eager version management to always store new values "in place," which means that no data move on commit (even if data has been replaced from a cache).
- I develop a means to extend a MESI directory protocol with *sticky states*, which enables (a) fast conflict detection on evicted blocks and (b) fast commit by lazily resetting the state of evicted blocks.

Because the log resides in virtual memory, LogTM can perform version management for transactions of any size. Hardware complexity is reduced because version management is decoupled from both processors' caches and the coherence protocol. Furthermore, the most complex component of version management, switching between old and new versions, is implemented in software. The common case, commit, is fast because memory is updated in place during transactions. Chapter 3 describes LogTM's API and general mechanisms for performing conflict detection and version management.

In Chapter 4, I discuss implementation tradeoffs in LogTM and present two implementations of LogTM's hardware conflict detection, LogTM-Dir and LogTM-Bcast. Both systems detect in-cache conflicts using an invalidation-based coherence protocol but do not require that transactional data be cached. LogTM-Dir extends directory protocols with *sticky states* to perform conflict detection even after transactional data has been replaced from caches. LogTM-Bcast performs the same task by adding Bloom filters [7] to broadcast coherence. For ease of implementation, LogTM always has the processor whose coherence request causes a conflict be the one that resolves the conflict by waiting (to reduce aborts) or aborting (if deadlock is possible).

In Chapter 5, I show that the scalability and performance of using LogTM transactions for synchronization on these systems are equal or better than those of lock-based synchronization. The remainder of Chapter 5 presents an evaluation of several of the implementation tradeoffs discussed in Chapter 4.

The evaluation in Chapter 5 demonstrates that LogTM is a viable approach for implementing transactional memory. However, the implementations described in Chapter 4 are limited in several important ways. In Chapter 6, I discuss several of those limitations and outline solutions. I discuss research related to LogTM in Chapter 7.

Chapter 2

Background

In this chapter, I provide background information and establish terminology used throughout this dissertation. First, I briefly describe the problem of cache coherence in shared-memory multiprocessors. Next, I discuss two alternative models for writing parallel programs on shared-memory multiprocessors, multithreaded programming with mutual exclusion and transactional programming. Finally, I describe previously proposed transactional memory systems and their limitations, which LogTM seeks to overcome.

2.1 Cache-Coherent Multiprocessors

This dissertation concerns the design and programming of cache-coherent shared-memory multiprocessors. Cache-coherent multiprocessing, which has been widely used in server systems for many years, is now ubiquitous as all high-performance microprocessors now include two or more processing cores per chip. Cache-coherent multiprocessors contain several independent processors that share a single memory system. To improve the effective access time of the shared memory, the individual processors maintain private caches. Special hardware keeps these caches *coherent*, ensuring that every load—whether it gathers its data from a cache or memory—returns the value established by the previous store to that memory location. Multiprocessors commonly enforce cache coherence using a broadcast snooping or directory-based coherence protocol.

In broadcast snooping [25] systems, all processors share a coherence bus (or other broadcast medium). When a processor attempts to access a block not present in its cache, it places a coherence request on the bus. The request may be satisfied either by the memory, or by another processor that has a valid copy of the block in its cache. All cache requests are ordered by the sequence on which they appear on the bus, and all processors see all coherence requests.

Directory-based coherence protocols [10] reduce inter-processor bandwidth by not broadcasting coherence requests. All coherence requests are sent to the directory, which tracks the presence of the block in all caches. The exact combinations of states and messages depends on the protocol and is the subject of ongoing research. The directory relays the coherence request by sending messages to other processors and to memory, if necessary. To avoid congestion in any one node, the directory is one node (possibly a single processor or group of processors) acts as the home node for each memory block.

A common class of directory-based coherence protocols are invalidation-based MESI protocols [78]. In a MESI protocol, each block of memory resides in one of four logical states: Modified (M) the block is valid in exactly one cache and not memory; Exclusive (E) the block is valid in one cache and memory; Shared (S) the block is valid to read in one or more caches; and Invalid (I) the block is not valid in any cache. These states are also used to track the state of memory blocks in processors' caches. A processor may read a block if it is in states M, E or S, but may only write a block if it is in state M. The home node tracks the state of the block in all caches. For example, some directories record for each block either the address of one processor that holds the block in E or M or a *sharers list*—a list of processors that contain a shared copy of the block. When a processor executes a load to a block not valid in its cache, it requests a shared copy of the block by sending a get-shared (GETS) request to the directory. Upon receiving the request, the directory either fetches the data from memory and adds the requestor to the sharers list, or forwards the request (FWD-GETS) to the processor that has exclusive access to the block. When a processor executes a store to a block not present in the M state in its cache, it sends a get-exclusive (GETX) request to the directory. The directory then sends invalidation messages (INV) to all processors on the sharers list, instructing them to invalidate their copy or the requested block or forwards the request (FWD-GETX) to the processor that holds an exclusive copy of the block.

2.2 Multi-Threaded Programming

One of the most common programming models for shared memory multiprocessors is multi-threaded programming, in which a program consists of multiple cooperating *threads* of control that share the same address space and some region of memory. In this model, a thread is an abstraction of the state of a processor. Logically, all threads execute simultaneously, sharing memory, but maintaining private processor and register state. Physically, however, the number of executing threads is limited by the number of processors available to the program. Because all threads share the same memory region and address space, there is no need for programmers to partition the program's data. In addition, threads can communicate through normal loads and stores. These conveniences and the widespread availability of shared-memory multiprocessors have contributed to the popularity of the multi-threaded programming model.

The challenge in this programming model is to coordinate accesses by multiple threads to shared objects. Consider one common data structure in multi-threaded programming, the task queue. Tasks can be placed on the queue at any time. The tasks are later executed by the next available (idle) thread. This strategy provides a simple mechanism for distributing work to a pool of a threads. Typically, each thread begins an iteration over its outer-most loop by grabbing an entry from a shared task queue. After reading the entry and removing it from the queue, the thread performs the task specified in the task queue entry. Because the task queue is shared, however, uncoordinated accesses to it by multiple threads could result in errors. For example, if the queue is implemented as a singly-linked list, threads might pull the next entry by: (1) reading the head pointer to a temporary variable, then (2) setting the head to the next entry (head->next). If two threads executed that same code simultaneously, however, both would read the head pointer and copy the address of the same task to their separate private variables. Next, both would set the shared head pointer to the next entry in the list. In the end, both threads would be set to perform the same task.

Dijkstra identified this as the mutual exclusion problem [19]. Threads sharing a memory pool alternatively execute *critical* and *non-critical* sections of program code. Only one thread at a time can be executing its critical section and a failure of a thread executing its non-critical section cannot cause other threads to block indefinitely. If the threads in the example above access the task queue only in critical sections, mutual exclusion is sufficient to ensure that each thread safely removes tasks from the queue without interfering with other threads.

Lamport generalizes the mutual exclusion problem by observing that interactions between critical sections are defined not by their actual overlap in physical time, but by causal relationships between them [43]. In order for any critical or non-critical section to causally affect another, the two sections must operate on at least one common memory location, and one of them must update that memory location. Thus, critical sections that do not share a causal relationship may execute concurrently without violating mutual exclusion.

2.3 Transactional Programming

An alternative to the multithreaded programming model described above is transactional programming. Transactional programming was first proposed in database systems as an intuitive model for application programmers to specify concurrency and durability requirements [27]. The transaction model combines the apparent serialization of critical sections with atomicity—the guarantee that a transaction will execute completely or not at all. Together, these properties guarantee that transactions will appear to execute in some serial order.

Transactions in software are analogous to the familiar concept of a business transaction. Two or more parties promise to take some action or set of actions contingent on the actions of the other parties. Either all parties fulfill their promises, or none do. Transactions in computer software work the same way; all the individual actions are completed or none are. More formally, a transaction in software is a transformation of state that is *atomic*, *consistent*, *isolated* and *durable*. Atomicity guarantees that each transaction will either complete all or none of its actions. Consistency requires that each transaction will find and leave the system in a consistent state (it is the burden of the programmer to ensure that each transaction if run independently will leave the system in a consistent state). Isolation guarantees that each transaction will execute as if it is the only transaction running in the entire system. Effectively, this means that transactions will appear to be executed serially in some global order. Durability guarantees that, once executed, the effects of a transaction will not be undone even if the system fails.

The terminology used to describe transactional memory operations was developed in the database community. A transaction *begin* starts the execution of a transaction—the transaction becomes *active*. An active transaction can either *commit*—complete successfully—or, *abort*—terminate before completion, discarding any updates. The *read set* of a transaction is the set of objects (or memory locations) read by the transaction. Similarly, the *write set* of a transaction is the set of objects modified by the transaction.

This programming model has been widely successful in the database community and today, database mangement systems are among the most scalable commercial applications.

2.3.1 Transactional Memory

Transactional memory is a programming model that applies the transaction concept to memory accesses in shared-memory programs. The semantics of memory transactions are the same as that of database transactions with one exception—memory transactions are not guaranteed to be durable (memory state itself is not typically durable across system failures). The lack of a durability requirement allows memory transactions to incur much less overhead than their database equivalents. In practice, memory transactions are often used as a replacement for mutual exclusion locking in critical sections. Consequently, memory transactions are typically much shorter (in number of instructions and execution time), access less memory and contain fewer I/O operations than database transactions.

2.4 Concurrency Control

Concurrency control is the method by which a particular parallel execution model is enforced. In essence, this amounts to the coordination of accesses by various concurrent threads or transactions sharing state. This section describes several concurrency control algorithms used to enforce mutual exclusion and transactional isolation.

2.4.1 Concurrency Control for Mutual Exclusion

Mutual exclusion is typically enforced through the use of *locks*. A lock is simply a memory word used to coordinate access to a particular data structure or critical section. Programmers often use a lock to *protect* a shared data structure. By convention, a thread must *acquire* the lock (e.g., by setting the lock word to '1') before it accesses the data and hold the lock for the duration of its critical section. In this convention, a lock can only be held by a single thread. While the lock is held, any other threads that attempt to acquire the lock must block. threads may block by *spinning*—repeatedly testing the status of the lock—to ensure immediate notification when the lock is released, or by suspending—allowing the processor to execute other threads—to save resources and possibly prevent deadlock. After a thread finishes its critical section, it *releases* the lock (e.g., by setting the lock word back to free). If all threads follow this convention, mutual exclusion is ensured because a thread must hold the lock throughout the critical section and only thread may hold the lock at any given time.

Locks are typically acquired using atomic read-modify-write instructions. These instructions allow a thread to atomically test the lock word (to see if any thread holds the lock) and set it (to acquire the lock).

2.4.2 Lock-Based Concurrency Control in Database Systems

Database systems have traditionally implemented transactions using a combination of locking and logging [26]. Database locks may be shared—allowing multiple concurrent readers—or exclusive—allowing a single writer. Database locks may be short or long. A *short lock* is a lock that is held for only one database operation. A *long lock* is a lock that is held from the time it is acquired until the end of the current transaction.

Database transactions will execute atomically if they are both *well-formed* and *two-phase*. A transaction is well-formed if it acquires at least a shared lock for each database object it reads and an exclusive lock for each object it modifies. This locking can be performed at any granularity so long as each active database object is covered by an appropriate lock. A transaction is two-phase if it acquires all of its locks before it releases any lock.

Most database systems enforce these requirements with *strict two-phase locking* and *write-ahead logging* [68]. In addition to being well-formed, strict two-phase locking requires that a process executing a transaction does not release any locks associated with that transaction until it completes (i.e. all locks are long locks). Write-ahead logging requires that the database record a log entry to stable storage for each database update before the effect of that update is recorded to stable storage. This policy ensures that the database has a record of, and the ability to undo, all updates from transactions that have not yet committed [26].

Although the transactional programming model requires that transactions appear to execute atomically, databases run many transactions in parallel in order to process transactions more quickly. Some transactions, however, *conflict* and cannot be safely overlapped. A transaction conflict occurs when two transactions access the same object and at least one of the accesses is an update (the same criteria Lamport required for a causal relationship). The transactions outlined in Figure 2-1 demonstrate a simple conflict. Both T0 and T1 access variables A and B, both of which are modified by one of the transactions.

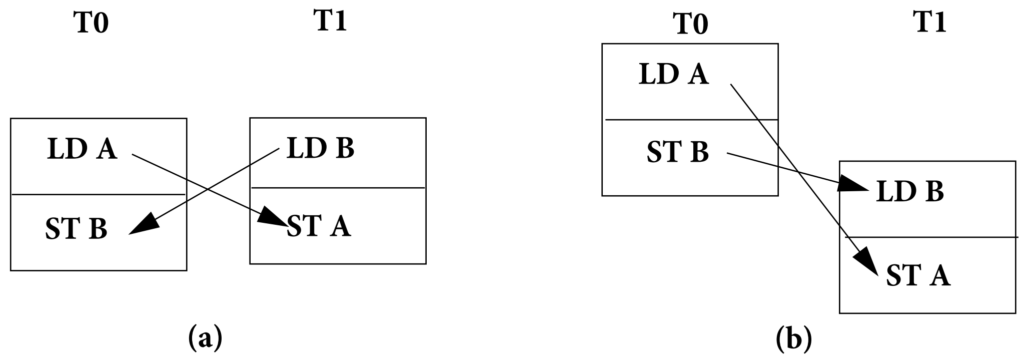


FIGURE 2-1. Conflicting Transactions: (a) An Illegal, Non-Serial Execution, and (b) A Serial Execution.

Since the two-phase requirement prevents a running transaction from surrendering any lock on an object that it has accessed, one transaction may have to wait until another processor commits or aborts its transaction before it can continue. This waiting can lead to deadlock if two transactions are each waiting for an object held by the other. More generally, deadlock can occur any time there is a cycle in the graph of dependences between transactions. For example, in Figure 2-1 (a), both transactions T0 and T1 successfully acquire the first block in their transaction, loading A and B respectively. Once they read these variables, however, they are not allowed to release the locks held on them for the duration of the transaction. Neither transaction can proceed until the other aborts. Database systems can avoid deadlocks by aborting transactions when they attempt to block, or detect deadlocks by tracking the dependences between active transactions. Most systems allow deadlocks to form, then resolve them by aborting one of the blocked transactions [68].

2.4.3 Optimistic Concurrency Control in Database Systems

An alternative to strict two-phase locking in database systems is *optimistic concurrency control* [42]. Optimistic concurrency control is motivated by the observation that locking introduces an overhead that is only necessary in the worst case. Instead of using locks to prevent conflicting accesses, optimistic concurrency control mechanisms detect conflicts and abort the offending transaction. Optimistic concurrency schemes divide transactions into three phases: read, validate and write. During the read phase, a transaction updates local copies of the modified objects. After a transaction completes its read phase, the system scans the read set to see if all the values read are still valid. A transaction T is valid unless its read set overlaps with the write set of a transaction that executed its write phase while T executed its read phase, or if T's write set overlaps the write set of another transaction that ran its write phase while T ran its write set. If T fails validation, it aborts without executing its write phase.

2.5 Logging and Recovery in Database Systems

Database systems that use lock-based concurrency control typically update tuples in place during transactions for efficiency. The atomicity requirements of transactions, however, require that, if a transaction aborts, all of its updates must be undone. Many systems implement crash recovery and transaction rollback by maintaining an undo/redo log. Using the popular ARIES recovery algorithm [52], transactions add entries to a shared log for each update they make. Each log record contains the before-image (old values) and after-image (new values) of the updated record and the *LastLSN*, the log sequence number of the most recent previous log record pertaining to its transaction. Starting at the end of the log, the rollback copies the old values from the log record back to their original location (the modified tuple) and follows the LastLSN to find the next most recent log record for the transaction. Once the last LSN field is null, the undo is complete; and, all of the transactions have been undone in reverse order.

2.6 Transactional Memory Systems

Several transactional memory systems have been proposed with varying levels of hardware support. This section describes some particularly influential systems and characterizes several classes of transactional memory system.

2.6.1 Hardware Support for Database Transactions

Some of the first proposals to add transactional semantics to the hardware interface sought to provide hardware support for the execution of database transactions. One such example is the 801 Storage system [13]. The 801 includes three architectural features designed to make database programs more efficient and easier to write: (1) large virtual storage, (2) *database storage* and (3) a 1-level store. Their innovative database storage includes support for database-style transactions on the one-level store with support for atomic commit, undo and recovery. Their programming interface makes storage to memory and the file system equivalent—any data structure supported in memory can be saved to disk. The transactions they envision are similar in scale to database transactions. They are long-running (thousands of instructions) and update stable storage [13].

2.6.2 Early Hardware Transactional Memory Systems

To the best of my knowledge, Knight's "An Architecture for Mostly Functional Languages" [40] is the first to propose transactional operations on memory. Knight reasons that the transaction model is a convenient interface between the compiler and the hardware to automatically extract parallelism from sequential programs written in "mostly-functional" languages. Unlike in purely-functional languages, some actions are allowed to have side effects (stores), and therefore must be performed in a particular order. Knight proposes that the compiler group operations into transactional blocks, which may include many loads, but only one store.

In Knight's scheme, the hardware executes transactional blocks optimistically in parallel except for their one store. Transactional blocks are "confirmed" (committed) in program order. The hardware maintains a dependency list for all memory locations read by each transaction. As transactions are confirmed, the hardware checks the dependency list of each active transaction. Any transaction which depends on the store from the newly-confirmed transaction is aborted and re-executed. Knight outlines an implementation based on two fully-associative caches. One cache, the dependency cache, tracks which main-memory locations have been read during the current transaction. The other, the confirm cache, holds transactional stores (although the transactional blocks in the paper are limited to only one store, the hardware would support transactions with as many stores as fit in the confirm cache). The data in the confirm cache is only written back to main memory when the transaction is confirmed [40].

Some processors provide the Load-Locked Store-Conditional (LL/SC) synchronization primitive [37, 15, 75], which allows atomic read-modify-write operations on a single memory location. LL-SC synchronization uses two separate instructions. The first, `load-locked` loads the value of a memory location into a processor register and begins to monitor the memory location for stores. The second, `store-conditional`, stores a value to the memory location only if that location has not been modified since the execution of the last load-locked instruction. LL/SC can be used to implement many common synchronization primitives such as test-and-set, fetch-and-increment/decrement, and compare and swap.

Stone et al. propose the Oklahoma Update Protocol [77], which extends the LL/SC primitive with the introduction of multiple reservations, or linked registers. The extra linked registers are combined with a different update mechanism, which allows multiple atomic stores. A "transaction" in this scheme proceeds as follows. First, the process performs one or more "Read-and-Reserve" operations, which define the read-set of the transaction. The process can speculatively update any of the locations previously read with a "Store-Contingent" operation. Finally, the process attempts to commit the transaction with a "Write-If-Reserved" action. The commit operation checks the valid bits of each reservation register, to see if any of the linked locations has been updated during the transaction. If all the reservations are still valid, the process enters the "commit phase." During the commit phase, coherence operations are deferred. The commit phase requires obtaining exclusive access to all memory locations updated by the transaction. Once the processor has obtained the proper permissions for each block updated in the transaction, all of the updates are propagated back to the cache, any deferred requests are processed and the transaction exits successfully.

2.6.3 Hardware Transactional Memory

Transactional Memory was first introduced by Herlihy and Moss introduced in their 1993 ISCA paper, "Transactional Memory: Architectural Support for Lock-Free Data Structures [34]." Their goal in designing a memory system based on atomic transactions was to support short critical sections—normally protected by locks—directly in hardware. Programmers, with the support of a compiler, designate transactional memory operations with special instructions.

Specifically, Herlihy & Moss propose the addition of four transactional instructions: *load transactional* (LT), *store transactional* (ST), *validate* and *commit* to the instruction set of a typical RISC architecture. LT and ST instructions signal to the processor that the data accessed by this load or store should be placed in the transactional cache. They also indicate that a transaction has begun if one is not already active. The validate instruction returns the status of the current transaction. The commit instruction returns a boolean value indicating whether or not the transaction succeeded (a violation of transactional semantics does not trigger a fault or trap). The programmer and compiler are free to choose the appropriate course of action in the case of a failed transaction.

They propose an implementation that uses a modified victim cache to store data accessed as part of a transaction. This transactional cache is the key mechanism in their scheme. All values accessed in a transaction are stored in a separate transactional cache. This transactional cache, they argue, should be small and fully-associative (very much like a victim cache [38]) so that the size of a transaction is not limited by worst-case cache conflicts, and so that transactional state can be cleared quickly (1 cycle in their proposal) in the case of a failed transaction. Since there is no requirement that the transactional cache hold only transactional data, unused entries in the transactional cache can be used as a victim cache for non-transactional data. Unmodified data stored in the transactional cache can be in any of the normal cache states. Transactional stores cause the allocation of two entries in the transactional cache: the original (non-modified) data word, and the new (updated) value. The copy that represents the value modified by the transaction will survive if the transaction commits and is marked with the transactional cache state XABORT. The other, which stores the value in memory before the transaction, will persist if the transaction aborts is marked with the state XCOMMIT. If a processor aborts its current transaction, it signals the transactional cache to invalidate all entries in the XABORT state and change all entries with state XCOMMIT to normal (non-transactional) states. At the end of a transaction, the executing processor attempts to verify that no other processors have made intervening accesses to a memory location involved in the transaction. If it detects a conflicting memory action, the transaction is aborted and none of its updates are propagated to main memory. Otherwise, the transaction succeeds and all of its operations appear to take place atomically.

The cache coherence protocol proposed is based on Goodman's snooping protocol [25] (the authors also develop an adaptation based on a directory protocol in a separate technical report [33]). Non-transactional requests behave exactly the same as in the Goodman protocol. However, requests for data may be refused by a processor in the middle of a transaction (with a "busy" response) as long as the request is also made as part of a transaction. The protocol, however, does not allow a busy response for non-transactional requests from other processors. The requesting processor, upon seeing the refusal, aborts its own transaction and retries (possibly waiting to avoid repeated conflicts). A side effect of this policy is that non-transactional operations take precedence over transactional ones because non-transactional requests are always granted even if the responding processor is actively executing a transaction. This policy also violates strict two-phase locking since a non-transactional request can "steal" a block of memory away from a processor in mid-transaction. The authors preserve transactional semantics by guaranteeing that the transaction in the example above will never commit.

2.6.4 Recently Proposed HTMs

The emergence of CMPs has reenergized research on transactional memory. Recently, several researchers have proposed HTMs, which differ primarily in their implementations of version management and conflict detection. Upon a store, some transactional memory systems use *eager version management* and put the new value in place. Others use *lazy version management* to (temporarily) leave the old value in place. Conflict detection is *eager* if conflicts are detected upon executing offending loads or stores and *lazy* if conflicts are detected later (e.g., when transactions commit). The taxonomy in Table 2-1 characterizes several recently proposed HTMs as to whether they use eager or lazy strategies for version management and conflict detection.

TCC. Hammond et al.'s *Transactional Memory Coherence and Consistency (TCC)* [29], like database systems that use optimistic concurrency control, uses lazy version management and lazy conflict detection. Store values are buffered at the processor's L1 cache and overwrite the L2 cache and memory only upon commit. TCC detects conflicts between transactions by broadcasting each transaction's write set on commit. Pending transactions abort if any memory location in the broadcast is a part of their write set.

LTM. Ananian et al.'s *Large Transactional Memory (LTM)* [5] uses lazy version management on cache overflows, storing new values in a hash table in uncached memory, while old values remain in place. When data fits in cache, LTM stores the new value in cache and the old value at memory. This coaxes the coherence protocol to store two different values at the same address. Repeated transactions that modify the same block, however, require a writeback of the block once per transaction. In contrast to TCC, LTM detects conflicts eagerly, checking for conflicts on each memory operation. LTM conflict detection is complex, however, due to support for the case where a processor replaces transactional data. Upon receiving a coherence request from the directory, such an LTM processor must walk an uncacheable in-memory hash table before responding (and possibly aborting).

		Version Management	
		Lazy	**Eager**
Conflict Detection	**Lazy**	TCC [29] (like OCC DBMSs [42])	
	Eager	LTM [5], VTM [66] (on cache conflicts)	UTM [5], LogTM [new] (like CCC DBMSs [21])

TABLE 2-1. A Transactional Memory Taxonomy

VTM. Rajwar et al.'s *Virtual Transactional Memory (VTM)* [66] also uses lazy version management on cache overflows with memory holding old values and an in-memory table (XADT) holding new (and a second copy of old) values. VTM does not specify version management when data fits in cache, but rather recommends other proposals [5, 29, 34, 65, 77]. VTM also uses eager conflict detection.

UTM. Ananian et al.'s *Unbounded Transactional Memory (UTM)* [5] proposes using both eager version management and eager conflict detection. This follows the example of the vast majority of DBMSs that use *conservative concurrency control (CCC)* [21]. UTM's implementation is complex, however, requiring substantial changes to the memory system.

Chapter 3

Log-Based Transactional Memory (LogTM)

LogTM is a strategy for implementing transactional memory a transactional memory system that combines software-based version management (with limited hardware support) and conservative hardware conflict detection to support arbitrary-sized transactions with limited hardware. LogTM adapts a well-known database algorithm for implementing transactions, strict two-phase locking and write-ahead logging. LogTM uses write-ahead logging to perform version management by saving old values before new values are written in place. LogTM detects conflicts eagerly, in a manner equivalent to strict two-phase locking. To balance implementation cost and performance, LogTM divides the work of providing atomicity and isolation in transactions between hardware and software. For speed, hardware detects conflicts and can aid logging; to save complexity, transaction updates are rolled back by software. The following sections describe the requirements for LogTM's version management, conflict detection and conflict resolution. The rest of the chapter describes the LogTM API and discusses the trade-offs involved in implementing the various components of transactional memory in hardware or software.

3.1 Eager Version Management

A defining feature of LogTM is its use of eager version management, wherein "new" values are stored in place while old values are saved in an alternate location. Specifically, in LogTM, the old values of registers are saved in a register checkpoint and old values of memory are stored in the *transaction log*, a thread-private section of the user program's virtual address space. If the transaction aborts, a software abort handler restores old values from the log and register checkpoint.

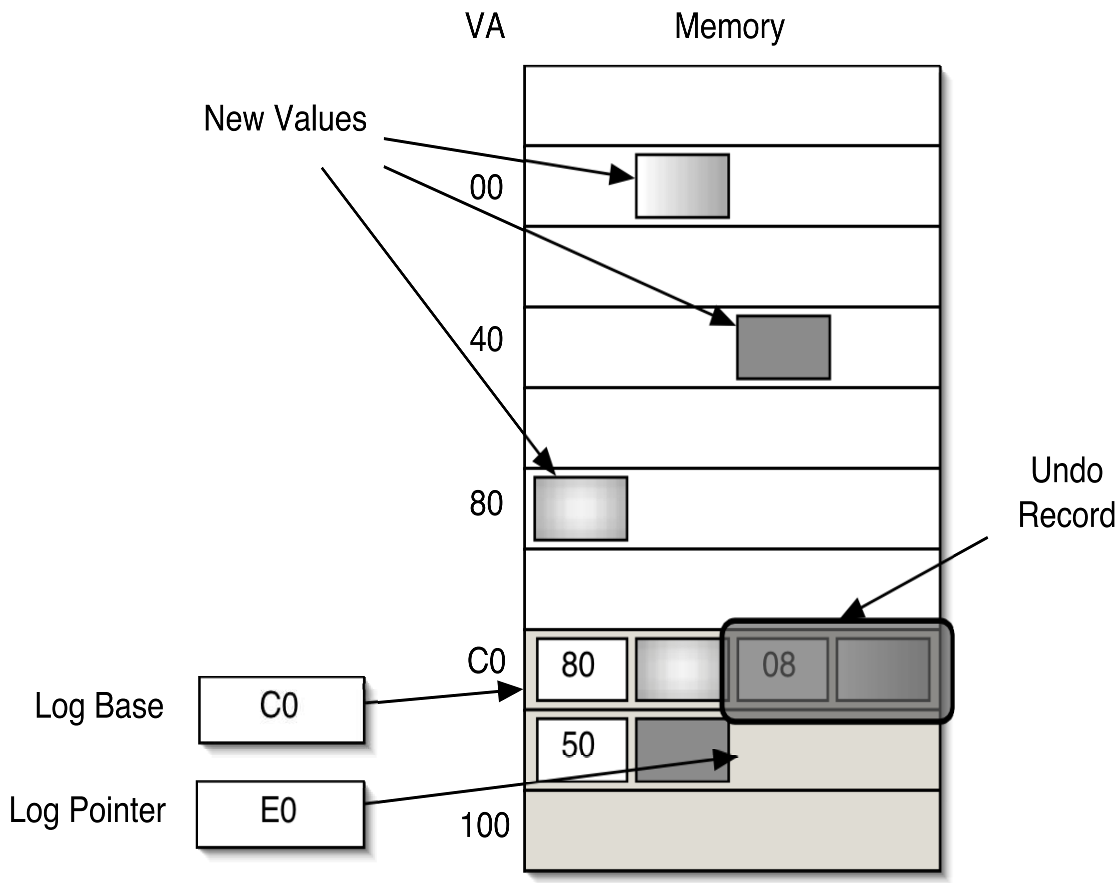


FIGURE 3-1. The Transaction Log.

3.1.1 The Transaction Log

The transaction log is a region of virtual memory, used by each thread to store the old values of all memory locations modified during a transaction. The base and bounds of the log are defined by two processor registers: `Log Base` and `Log Pointer`. A thread sets `Log Base` when it allocates space for the log—e.g., at thread creation, or when the LogTM system is initialized. The LogTM system updates `Log Pointer` each time it adds an entry to the log. Figure 3-1 illustrates the layout of the log in virtual memory. The main array is a representation of virtual memory, with virtual addresses shown on the left. The shaded area (bottom of Figure 3-1) represents the transaction log for the current thread. The unshaded area represents shared memory locations, e.g., part of the heap.

A LogTM system maintains the log at a fixed granularity, logically dividing memory into fixed size blocks. Figure 3-1 shows a region of shared memory (unshaded) and the log of an active transaction that has executed three stores (shaded). The three pairs of shaded blocks in Figure 3-1 represent the old and new values of three memory blocks, each 8 bytes long. The new value of each block is stored in place, and the old value is stored in an undo record in the log. Each undo record includes the virtual address of the modified block and its old value. Because the purpose of the log is to restore pre-transaction values, a LogTM system need only log the first update to any given memory location.

In database terminology, the transaction log is an undo-only log—i.e., it does not contain sufficient information to re-execute a transaction nor to recover the state of memory in the event of a crash [52]. This is in contrast to the logging used in most database systems. In a typical database, the log is used for recovery as well as transaction rollback. Importantly, because the transaction log is not used for recovery, there is no need to store the log on disk or other stable storage. Furthermore, since the log is not needed after the commit of a transaction, unlike the log in most database systems, the transaction log in LogTM is thread private and may be discarded after a transaction completes.

The log is defined in virtual memory for which physical memory is allocated on demand. If adding a log entry exhausts the physical memory in the log, the current thread takes a page fault and uses the existing virtual memory support to allocate an additional page for the log. Because the transaction log is stored in virtual memory, it may grow arbitrarily large. Log pages may be swapped to disk without impacting LogTM's version management.

3.1.2 Transaction Commit

On commit, the executing thread simply clears its transaction log by setting `Log Pointer` equal to the `Log Base` and discards its register checkpoint. No copying is needed because new values are already in place.

3.1.3 Transaction Abort

To maintain atomicity in the event of a transaction abort, LogTM must undo any updates from the aborting transaction by restoring the old values maintained in the transaction log and register checkpoint to their original locations in memory and processor registers. In order to save hardware complexity, LogTM performs this restoration in software. In LogTM, a transactional memory program registers an abort handler which will be called when a transaction aborts. The abort handler is expected to undo the effects of the aborted transaction using the transaction log. For example, a handler can rollback a transaction by reading the undo records in the log in last-in-first-out order, copying the old values in each record to the corresponding address. Once the handler has processed the log, it can restore the processor registers to their pre-transaction states.

3.2 Eager Conflict Detection

LogTM requires eager conflict detection, which means that a LogTM system must detect and resolve any conflict triggered by a memory request before that request completes. In LogTM, this detection is the responsibility of hardware. Implementations of LogTM are expected to leverage the coherence mechanism to implement conflict detection efficiently. LogTM implementations must report all true conflicts between concurrent transactions, but to reduce hardware complexity and cost, they are allowed to report false conflicts to simplify implementation.

3.2.1 Requirements

LogTM's eager conflict detection is based on strict two-phase locking. In place of the shared and exclusive locks used in database systems, however, LogTM requires read and write *isolation*. Unlike locking, isolation does not prescribe any particular conflict resolution mechanism (e.g., blocking). If a block is read isolated, it cannot be written by any thread without generating a conflict. If a block is write isolated, it cannot be read or written by any thread without generating a conflict.

Requirement 1: Transactions Must be Well Formed. In order for a thread to read a memory location in a transaction, that thread must first obtain read isolation on that location. In order to write a location, a thread must first obtain write isolation on that location. If an attempt to acquire read or write isolation results in a transaction conflict, the system must signal a conflict before the offending memory instruction is retired.

Requirement 2: Isolation Must be Strict Two-Phase. Any memory location that becomes read or write isolated by being read or written in a transaction must remain isolated until the commit or abort of that transaction.

Requirement 3: Isolation Must be Released at Transaction End. Conflicts may prevent one or more transactions from making forward progress. In order to ensure forward progress in the system, a thread must release its isolation when it aborts or commits its transaction.

3.2.2 Example Implementation

For small transactions—the expected common case—the entire read and write set of the transaction remains in the private cache of the executing processor. In this case, a standard invalidation-based cache coherence protocol is well suited to detecting transaction conflicts. As discussed in Chapter 2, cache-coherent multiprocessors that use invalidation-based coherence protocols typically enforce the invariant that every block of memory resides in one of three logical states: (I) the block is invalid in all caches, (S) one or more caches hold a valid read-only copy of the block and (M) one cache only has a writable copy of the block. Such systems already require that a processor obtain shared (read only) access to a memory location before it may be read and exclusive access (read/write permission) before it may be written. As a result, if isolation is provided by the coherence mechanism, these protocols enforce the well formed requirement by default. Additionally, these rules ensure that any memory access that hits in the local (private) cache will not trigger a transaction conflict.

Conflict detection in such a system works might work as follows: each processor tracks the read and write sets of its active transaction with two additional bits per line in its private cache. The read (R) bit indicates that the block has been read during the current transaction. The write (W) bit indicates that the block has been written during the current transaction and potentially contains data values from an uncommitted transaction. When a processor P executes a load or store, it first checks its local cache. If the corresponding block is not present, P issues a request for the block to the memory system. That request is sent to one or more other processors (e.g., via a broadcast or forwarded by a directory). Those processors check their local state (in the cache or memory controller) to detect conflicts with any transactions running there. The presence of a conflict is returned along with the coherence response. That response signals the conflict (if any) to P, which resolves the conflict. The states of any copies of the contended block in the responding processors' caches remain the same, as do the R and W bits there, until the transactions running on those processors end. This fulfills the strict two-phase requirement. Finally, at the end of a transaction, the processor flash-clears the R and W bits in its cache, fulfilling the requirement to release all read and write isolation after the transaction commits or aborts.

3.3 Conflict Resolution

Detecting a transaction conflict informs a transactional memory system that its current execution schedule will violate the isolation of one or more concurrent transactions. To preserve isolation, the system must *resolve* transaction conflicts by serializing the conflicting transactions. A transactional memory system can serialize the execution of a set of transactions by aborting or stalling one or more of the transactions. Resolving conflicts by aborting transactions risks livelock if all transactions are aborted before completion. Resolving conflicts by stalling, however, risks deadlock if transactions are forced to wait indefinitely.

In LogTM, transaction conflicts are resolved by the thread that makes the memory request that first causes the conflict. The requesting thread can resolve the conflict by aborting its transaction. Or, because the conflict is detected before the conflicting memory access is completed, the requesting thread can resolve the conflict by either stalling—waiting for the conflicting transaction to commit or abort .Because aborts are costly in LogTM, resolving conflicts via stalling can improve performance. Implementations that resolve conflicts by stalling, however, must take care to avoid deadlocks.

3.4 LogTM API

Table 3-1 presents LogTM's interface in three levels. The *user interface* (top) allows user threads to *begin*, *commit* and *abort* transactions. Compilers could translate higher level constructs such as an atomic block to LogTM's `begin_transaction` and `commit_transaction` calls like the Java compiler generates monitor enter and exit calls from statically scoped synchronized blocks [47]. The system/library interface (middle) lets thread packages initialize per-thread logs and register an abort handler. Upon an abort, LogTM lets the abort handler "undo" the log via a sequence of calls using the low-level interface (bottom). In the common case, the handler can restart the transaction with user-visible register and memory state

User Interface
`begin_transaction()` Increments TM Count. If TM Count > 0, subsequent dynamic statements form a transaction. Logically saves a copy of user-visible non-memory thread state (i.e., architectural registers, condition codes, etc.).
`commit_transaction()` Decrements TM Count. When TM Count == 1, commit ends a successful transaction. Discards any transaction state saved for potential abort.
`abort_transaction()` Transfers control to a previously-registered abort handler which should undo and discard the current transaction and set TM Count to 0. A system may choose to restart the transaction or execute other code.
System/Library Interface
`initialize_logtm_transactions(Thread* thread_struct, Address log_base, Address log_bound)` Initiates a thread's transactional support, including allocating virtual address space for a thread's log. As for each thread's stack, page table entries and physical memory may be allocated on demand and the thread fails if it exceeds the large, but finite log size. (Other options are possible if they prove necessary.) We expect this call to wrapped with a user-level thread initiation call (e.g., for P-Threads).
`register_abort_handler(void (*) abort_handler)` Registers a function to be called if a transaction is aborted. Abort handlers are registered on a per-thread basis. The registered handler should assume the following pre-conditions and ensure the following post-conditions:
Abort Handler Pre-conditions: Abort has occurred. System may have restored some or all memory blocks written by the thread to their pre-transaction state. Other memory blocks written by the thread (a) have new values in (virtual) memory but these blocks are isolated and (b) have their (virtual) address and pre-write values in the log. If a block is logged more than once, its first entry pushed on the log must contain its pre-transaction value. Log also contains a record of pre-transaction user-visible non-memory thread state.
Abort Handler Post-conditions: Abort handler called `undo_log_entry()` to pop off every log entry. Abort handler then called `complete_abort_with_restart()` or `complete_abort_without_restart()`.
Low-Level Interface
`undo_log_entry()` Reads a block's (virtual) address and pre-write data from the last log entry, writes the data to the address, and pops the entry off of the log. The system may end isolation on the block if is sure that pre-transaction value is now restored (i.e., there are not earlier duplicate log entries for this address).
`complete_abort_with_restart()` End isolation on all memory blocks, restore thread's non-memory state from last `begin_transaction()`, and resume execution there.
`complete_abort_without_restart()` End isolation on all memory blocks, discard thread's non-memory state from `begin_transaction()`, and return to abort handler. Use to handle error conditions.

TABLE 3-1. The LogTM Interface

restored to their pre-transactions values. Rather than just restart, an abort handler can also complete an abort and run arbitrary user code to manage aborts.

3.5 Example

Although presented separately in this chapter, conflict detection and version management interact in important ways. To better understand the way LogTM's eager version management and eager conflict detection work together, consider the example depicted in Figure 3-2. Figure 3-2 illustrates the logical execution of a simple transaction. Assume that the current thread's log begins at virtual address (`VA`) 1000 (all numbers in hexadecimal), but is empty (`Log Pointer=Log Base`). In this example, logging is performed on 8-byte blocks (the values of which data are given as a two-digit values) and conflict detection is performed on 16-byte cache lines (pairs of blocks). The R and W flags on the right indicate whether a cache line is read or write isolated by the conflict detection mechanism. Circles indicate changes from the previous snapshot.

Part (a) shows the thread beginning a transaction by incrementing its `TM Count`. Part (b) shows a load from virtual address 00 acquiring read isolation on that block. Part (c) depicts a store to virtual address 40 acquiring write isolation on that block and logging the block's virtual address and old data (34). Part (d) shows a read-modify write of address 22 that acquires read and write isolation for the encompassing block and writes the log with the block's virtual address and old data (23). Part (e) shows a transaction commit that resets `TM Count` and `Log Pointer`, and releases all read and write isolation. Part (f) shows an alternative where instead of committing after part (d), the thread aborts its transaction and must restore values from the log before resetting `TM Count` and `Log Pointer`, and releasing read and write isolation.

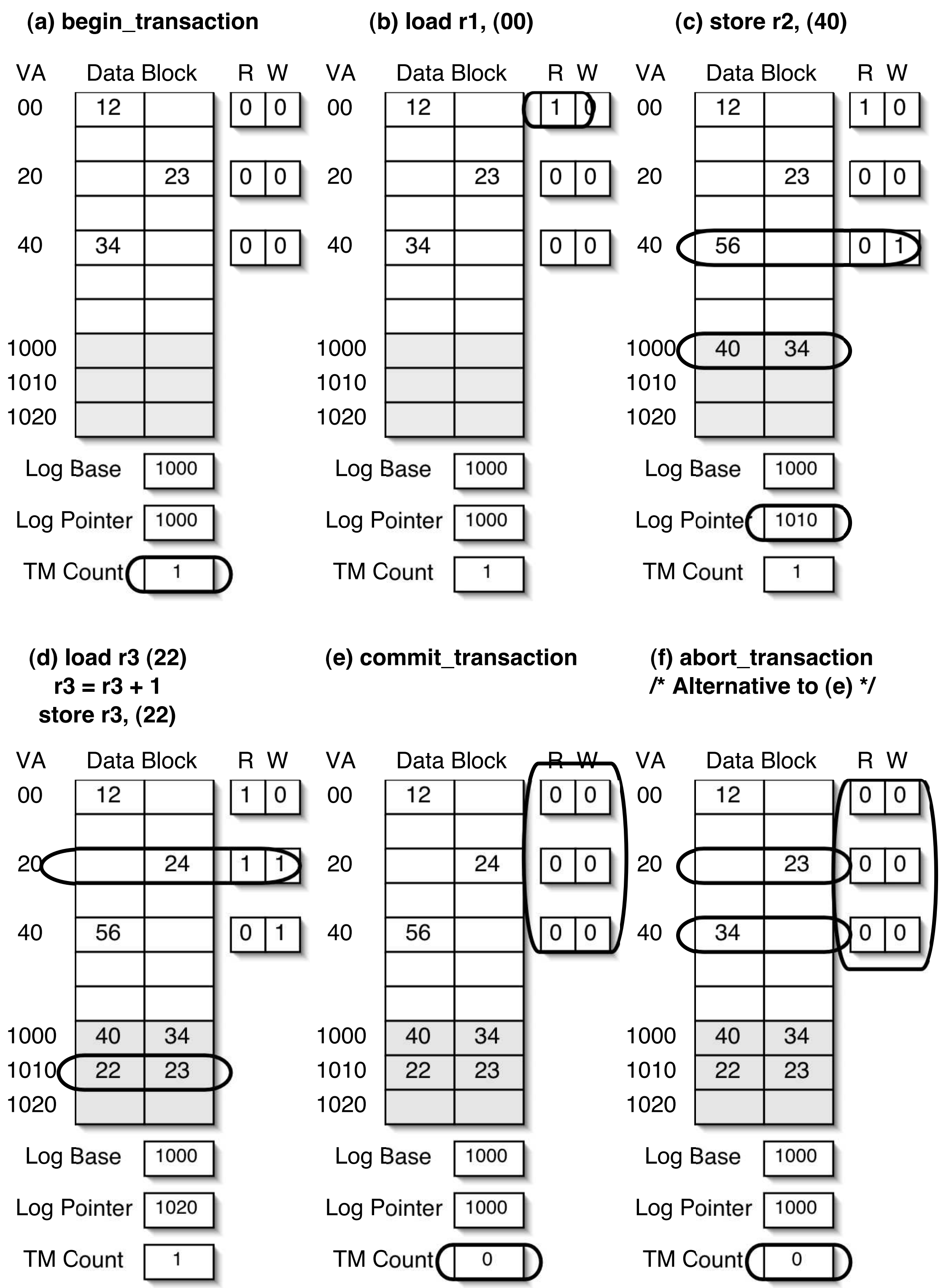


FIGURE 3-2. Execution of a Transaction with Two Alternative Endings.

3.6 Discussion

LogTM is designed to provide robust performance when transactions exceed the size or associativity limits of the hardware. To support such transactions, LogTM must therefore provide both conflict detection and version management for data outside processors' private caches and other dedicated hardware structures. Conflict detection and version management pose separate challenges to operating outside the cache. The distinct characteristics of these challenges lead LogTM to handle each differently, extending hardware to provide conflict detection for out-of-cache transactions and deferring version management to software.

Version management in unbounded transactions is difficult because the space required to store the separate versions is unbounded. Version management is the maintenance of the program's data values and must therefore be performed precisely—i.e., values cannot be altered, lost or associated with the wrong version. Because this version information must be maintained precisely, a transactional memory system must provide storage for both old and new versions of each object modified in a transaction, which requires space equal to the write set of the largest transaction in the program (in addition to the program's existing space requirements). Memory provides ample space to maintain both old and new versions of transactional data, but using memory in such a way requires associating separate addresses for each version and implementing a policy to ensure that every memory access reaches the proper version. Fortunately, however, versions are switched rarely—on abort if eager version management is used, or on commit otherwise. LogTM employs eager version management because aborts should be rarer than commits. LogTM takes advantage of that rarity by passing the responsibility for switching versions to software, in the form of a software abort handler.

Conflict detection for unbounded transactions is challenging because, unlike version management, it must be logically performed on every memory access by every processor. Although caches can filter out conflict checks for accesses that hit in the cache, conflict detection is still performed frequently. Conflict detection is especially challenging to implement in software because it requires remote operations, i.e., the read and write set of an overflowing transaction must be checked on every cache miss from every other processor. Fortunately, unlike version management, conflict detection need not be performed precisely, but merely *conservatively.* A conflict detection mechanism must report all true conflicts between transactions, but reporting a conflict when one does not exist (false conflict) will affect performance, but not correctness. LogTM leverages this property by tracking read and write sets conservatively in hardware. By allowing false conflicts, LogTM can detect all true conflicts between transactions of any size while using finite structures—amenable to hardware implementation—to track read and write sets.

Most HTMs that support large transactions proposed thus far have used lazy version management [5, 11, 29, 66]. Doing so allows these systems to abort transactions quickly and easily. Because aborts are fast and simple, these systems can choose to abort transactions when they encounter traps, interrupts or other operating system events that interfere with transaction processing. LogTM, however, strives to support large and long-running transactions. To do so, it must handle such events without relying on aborting transactions.

Eager version management provides two important advantages in LogTM. First, updating memory in place means that loads never need to check local write buffers. Second, using eager version management makes commits fast even for large transactions. Ideally, transaction commits will be more common than transaction aborts. Early studies suggest that this is likely to be the case for most workloads [5, 16, 54, 67]. Furthermore, resolving conflicts by stalling rather than aborting (Section 4.3) and *write set prediction* (Section 4.3.1), can significantly increase the fraction of transactions that commit.

Eager conflict detection detects conflicts sooner and eliminates the need for cascading rollbacks on systems that use eager version management. Detecting conflicts early can reduce wasted work in two important ways. First, detecting a conflict early allows a transactional memory to abort a transaction early, giving it the opportunity to switch execution to a different thread or transaction. Because transactions are atomic, any transaction that does not commit does not contribute to the progress of the program. A transactional memory system that can identify unsuccessful transactions early can abort those transactions sooner and potentially run other code that will make progress. Second, detecting conflicts early, allows a transactional memory system to resolve many conflicts by stalling instead of aborting a transaction, which eliminates wasting work all together.

In summary, strict two-phase locking and write-ahead logging has proven to be an effective strategy for implementing transactions in database systems. Eliminating the requirement for durability and leveraging hardware support allows LogTM to adapt this successful algorithm to efficiently implement lightweight memory transactions with little overhead. Eagerly making updates in place allows LogTM to support large transactions without copying on commit and makes processing commits, which should be more common, easier than processing aborts. Detecting conflicts completely in hardware allows for efficient execution even in presence of large transactions. This combination allows LogTM systems to balance performance and implementation cost in transactional memory.

Chapter 4

Implementing LogTM

In the previous chapter, I outline the LogTM system and API without specifying a particular implementation. Here, I discuss the challenges of implementing LogTM especially LogTM's eager version management and eager conflict detection. Section 4.1 discusses several trade-offs in implementing LogTM's version management and presents three solutions that vary in complexity and performance. Section 4.2 presents two concrete implementations of LogTM's eager conflict detection based on directory and broadcast-based coherence. Section 4.3 presents a policy for resolving conflicts in LogTM.

4.1 Implementing LogTM's Eager Version Management

Recall from the previous chapter that the LogTM interface (Section 3.4) specifies that the transaction log must contain an undo record for each block modified by the transaction at the time of an abort. LogTM implementations must also restore the value of processor registers to their pre-transaction values before a transaction can safely be re-executed. Implementations, however, are free to perform these tasks in many ways.

4.1.1 Implementation Trade-Offs

This sub-section describes several design dimensions LogTM system designers must address when implementing LogTM's version management.

Hardware vs. Software Register Checkpointing. LogTM designers seeking the highest performing system should consider including a hardware register checkpointing mechanism. Hardware register checkpointing has been proposed both for transactional memory [5, 29, 46, 65] and as a mechanism for implementing speculation in out-of-order processors [3]. Systems that use register renaming [86] can checkpoint the register rename table and lock physical registers holding transactional values. Systems that use architectural register files could employ a bulk copy mechanism such as the one used by the UltraSPARCIII processor [36].

Alternatively, to reduce hardware costs, register state could be saved and restored in software. The log provides convenient storage for register values. For small transactions that do not make any procedure calls, a compiler could be modified to generate code to save only the registers that will be overwritten by the transaction.

Implicit vs. Explicit Logging. Perhaps the most important design choice is whether logging will be performed *explicitly*—as specifically directed by software, e.g., a hardware instruction—or *implicitly*—as a side effect of another instruction, e.g., a transactional store. This decision is particularly important because it affects the instruction set architecture, which changes much less frequently than the microarchitecture. Explicit logging could be performed using existing instructions, e.g., by copying values from updated memory locations to the log in software, then incrementing the log pointer register directly. Explicit logging could also be enhanced by special instructions that use dedicated hardware to more efficiently create log entries. Such instructions could also optionally check hardware state that tracks transaction write sets to eliminate unnecessary duplicate log entries.

Implicit logging will likely be more difficult to implement than explicit logging. A store with implicit logging must: (1) translate the address of the log pointer; (2) read the old values from the target memory block and write them to the log, (3) record the target memory block's virtual address to the log, (4) write new values into the cache or store buffer, and (5) increment the log pointer. This is especially true for RISC architectures, in which instructions tend to be short, simple operations that can be performed directly by the microarchitecture. Modern micro architectures that convert CISC instructions into series of micro-ops

[35], however, provide an interesting alternative. Such machines could implement logging that is implicit as seen by application software, but implemented by explicit micro-ops. Stores in a transaction might translate to a different series of micro-ops than non-transactional stores.

Implicit logging, however, allows for a cleaner interface for transactional memory programs. Fewer additional instructions must be added to the instruction set, and functions that are called both inside and outside transactions do not need to be written or compiled differently for the two cases. Whereas, if explicit logging is used, any function that is invoked from within a transaction must contain these logging instructions.

Buffered vs. Direct Logging. LogTM requires that the transaction log be appropriately filled on abort. Prior to an abort, however, implementations are free to store undo information in any form. An implementation of LogTM could use a small *log buffer* to temporarily store a small number of log entries. The log buffer could be filled by hardware via either explicit or implicit logging. For explicit logging, hardware might provide a special log instruction that copies a memory block to the log buffer. Because the size of the buffer is limited, for large transactions, log entries in the log buffer must be spilled to the log itself (in virtual memory). These spills could be performed implicitly in the background, or explicitly by making the contents of the log buffer visible to software.

Use of a log buffer has three primary advantages. First, it can reduce bandwidth demand on the memory system. Because transactions often have small write sets, a small log buffer can eliminate the need for copying data to the log for many transactions. Second, storing to a log buffer could be used to reduce the pressure to translate log addresses. Because the log buffer is not part of virtual memory, address translation can be deferred until the buffer is spilled to memory. Finally, using a log buffer offers an opportunity to provide hardware support for logging without defining the log format in hardware. Instead, the log format can be defined by the software used to spill the log buffer to the transaction log in memory.

Logging Granularity. Logically, the log can be stored at any granularity provided that all transactional updates are included in the log, and that isolation is maintained on all memory locations that are logged. That freedom allows system designers to select the granularity that will perform best for their system and workloads. Using a larger granularity reduces the number of log records generated by each transaction. Using larger records, can potentially reduce the overall size of the log by reducing the number of addresses in the log, provided there is enough spatial locality in transactional stores to fill the larger records. Smaller records, however, can reduce the size of the log if more memory locations are logged than actually updated. Reducing the size of the log may reduce the amount memory system traffic added by logging.

Logging Location. LogTM systems are free to store log values—either buffered or in memory—at any level of the memory hierarchy. In general, memory that is closest to the processor (e.g., registers and L1 caches) is faster, smaller and more centralized. Higher levels of the memory system (L2/L3 caches and main memory) are larger, slower and more distributed. Storing log records near the processor generates additional memory system read traffic when data are modified, but not read, in a transaction. But, storing log records farther from the processor can generate additional memory system write traffic when the storage for a log record is not co-located with the corresponding store target (e.g., if they are stored in different cache banks). Also, because log entries are only read on transaction aborts, which are rare (Section 5.4), storing them in lower levels of cache (and not in the L1) may improve the effectiveness of the L1 cache for many workloads. In deciding where to store the log, there is a trade-off between using extra valuable capacity and bandwidth at low levels of the memory system, and managing the complexity of storing the log at lower levels of the memory system, where updated memory locations are potentially far from the memory that holds the log.

Most processors today include write back second-level caches protected by Error Correcting Codes (ECC) [61] for reliability. Such caches can be extended to create log entries. Store instructions typically modify a small number of bytes of memory, e.g., 4-8 B. Cache lines are typically much larger (e.g., 64-128 B). This discrepancy means that store operations typically merge new values—recorded by the given store instruction—with values already in memory. To perform this merge, the cache must hold a valid copy of the memory block. That copy contains the exact values that must be recorded in the undo record that corresponds to the store. Furthermore, if the cache is protected by ECC, the cache must read the values in the entire ECC word to compute a new ECC code after each such merge. Since the entire memory word is already being read, the calculation of the new ECC code provides a convenient opportunity to copy old values to the log, particularly if the log is maintained at the granularity of an ECC word.

Logging at higher levels of the memory system (e.g., second or third-level caches), although efficient in terms of memory system traffic, requires additional hardware complexity. First, L2 and L3 caches are typically physically tagged. Undo records, however, require the blocks' virtual address. Therefore, the processor must pass both the virtual and physical addresses of the target memory location. Second, because larger caches are typically multi-banked, the log target and store target may be stored in different banks. Therefore, values copied from the store target must be moved from their original bank to the bank that holds the corresponding undo record. Finally, although log stores are nearly always cache hits, the cache must be able to handle misses to the log (i.e., the memory block that will hold the undo record is not present). In that case, the cache must have sufficient space to buffer log values, or must be able to stall the processor until the miss has been serviced.

4.1.2 Compiler-Supported Software Logging

Designers seeking to implement LogTM with minimal hardware support should consider software-implemented explicit logging and software register checkpoints. The `Log Base` and `Log Pointer` registers can be read and modified by user instructions. Because the log is stored in user-addressable virtual memory, ordinary loads and stores can access log values as well. In software logging, user software generates a log entry by first writing the virtual address of the block to the log—the current value of

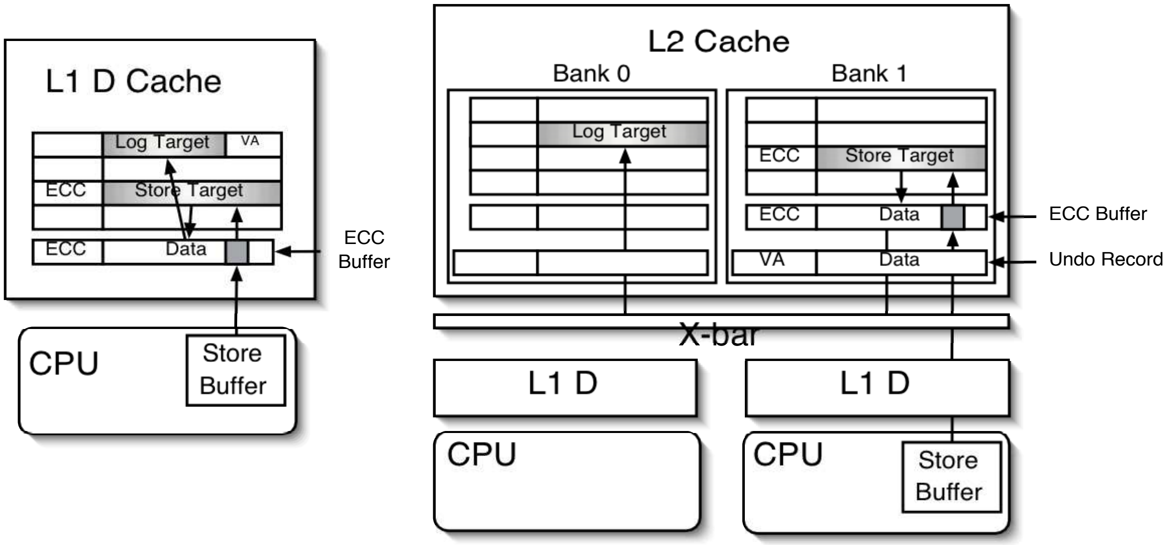
`Log Pointer`—then copying each word in the block to subsequent locations in the log. Once the log holds the block address and old values, the software increments the log pointer by the size of a log entry. To reduce the burden on the programmer, a modified compiler generates the logging instructions automatically. Because both the logging actions and the transaction rollback mechanism are under software control, the programmer can choose the logging granularity that is most appropriate for a given application.

Software logging instructions can be automatically generated by a compiler. The compiler's code generator could naively insert logging instructions that copy the appropriate memory block to the log before each store instruction. Functions called both from sites within transactions and outside transactions would have to be compiled twice, once with logging and once without. A runtime system using *just-in-time* compilation, however, could dynamically re-compile functions so that they include logging [1] on demand when they are called within transactions.

4.1.3 In-Cache Hardware Logging

To implement LogTM with the best possible performance, designers should consider adding hardware support for direct implicit logging at the first level of cache that is write-back. In such a system, the processor translates the target address and `Log Pointer` value for each transactional store into physical addresses and sends the physical addresses to the cache along with the new value. The processor uses a single-entry micro-TLB, such as the one used to pre-translate instruction pages in the VAX-11/780 [70], to translate `Log Pointer`.

If the L1 cache is write-back, write-allocate, and protected with ECC bits, as in the AMD Opteron [39], the L1 cache controller should create log entries at the same granularity at which ECC codes are computed. Figure 4-1 illustrates a LogTM system performing logging in an ECC-protected, write-back L1 cache. First, the processor sends `Log Pointer`, the new value, and both the virtual and physical addresses of the store target to the data cache. The cache translates the address of the log target (`Log Pointer`) using



(a) (b)

FIGURE 4-1. In-Cache Hardware Logging. (a) Logging in the L1 Cache, (b) Logging in the L2 Cache.

a micro-TLB. The cache then looks up both the store target and log target addresses (L1 cache lookups are often performed in parallel with address translation using virtual addresses and virtually indexed caches). Assuming both the log target and store target lookups hit in the cache, the cache next copies the value of the store target block (one ECC word) to a buffer (the ECC buffer in Figure 4-1) and immediately copies the value of that block to the log target. The cache can then append the store target virtual address to the old data values in the log to complete the undo record. In parallel, the cache merges the new value from the processor into the ECC word in the ECC buffer. When values are merged, the cache can calculate the ECC code for the modified block. Once the new code is ready, the cache copies it along with the entire ECC word back to the store target memory location.

If, on the other hand, the L1 cache is write-through, write-no-allocate and not ECC protected, like the L1 caches in the Sun UltraSPARC T1 (Niagara) [41], logging should be performed at the L2 cache, or the closest write-back cache. The Niagara already sends store values and physical addresses directly to the L2 cache, which is a multi-banked cache that is shared by the eight processors on each chip. The transaction log may be safely stored in a shared cache because each thread's log is private. In Niagara, even if the target line is present in the L1 cache, the L1 copy of the line is not updated until after the data reaches the L2. Figure 4-1 (b) illustrates the data flow in a LogTM system performing logging in Niagara-like caches (assuming both the log target and store target hit in the cache). Again, the processor sends `Log Pointer`, the new value, and the virtual and physical addresses of the store target to the cache. Here, the processor sends these values straight to the shared L2 cache using the on-chip crossbar (shown as X-bar in Figure 4-1 (b)). Specifically, the processor sends these values to the cache bank that holds the store target. The target bank copies old values from the store target memory location to the ECC buffer. Next, the bank copies these values again, and combines them with the virtual address of the store target, to form the undo record. The cache bank buffers the undo record while it arbitrates for the crossbar, which it uses to send the undo record to the cache bank that holds the log target. Next, the new values from the processor (shown as the shaded section in the Data field in Figure 4-1) are incorporated into the values in the register. One the full block is assembled, the new ECC code is calculated. When the new values and new ECC code are ready, the cache copies them to the target memory block.

Performing logging in the cache will reduce the traffic on the L1-to-processor interconnect. Logging in the L2 will reduce traffic to the L1 cache banks by eliminating the transfer of memory locations that are modified, but not read by the transaction. Logging in the L2, however, will increase traffic on the L2 bank interconnect whenever the target of a store and its corresponding log storage are not located in the same bank. In a system like the Niagara, Logging in the L2 will also complicate arbitration for the L2 crossbar.

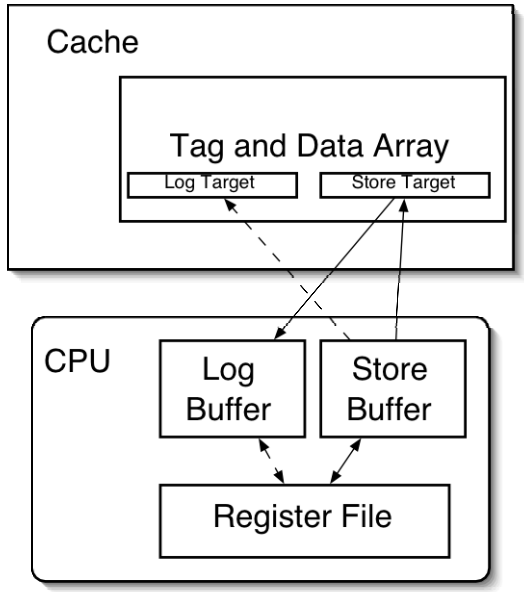


FIGURE 4-2. Hardware/Software Hybrid Buffered Logging.

4.1.4 Hardware/Software Hybrid Logging

An alternative to the complexity of all-hardware logging and the overheads of software-only logging is to support implicit logging with a log buffer located near the processor or L1 cache. The log buffer is filled quickly by hardware, and spilled periodically to memory by a software handler. Because the log buffer may be filled without translating the `Log Pointer` to the physical address of the log, logging using a buffer does not require additional address translation. Figure 4-2 depicts the log buffer and information flow in buffered logging. Solid arrows represent values sent during transaction execution (buffer fill) and dashed arrows represent values sent during buffer spill. Stores send new values to the store buffer and eventually to the cache (solid arrows). When the cache receives a store, it first sends the old values to the log buffer. When the log buffer fills, the processor takes a trap to empty the buffer to memory, much like TLB miss traps in SPARC processors [82]. As depicted by the dashed arrows, the trap handler reads old values from the log buffer first into registers, then writes them to memory with conventional store instructions.

Log buffers could provide the speed of all-hardware logging for many transactions without requiring the added complexity of (logically) performing an additional address translation for each transactional store instruction. Characterizations of critical sections in multi-threaded programs [16] and transactional memory workloads [67] as well as my experiments (Chapter 5), show that a small log buffer could process all logging for many transactions in the benchmarks studied in this dissertation.

4.2 Implementing Eager Conflict Detection

LogTM requires that implementations detect conflicts eagerly in hardware. Fortunately, existing coherence mechanisms are well-suited to detect transaction conflicts provided that transactional data reside in the cache. LogTM, however, requires that hardware detect conflicts for all transactions, even those with read and write sets that exceed the capacity or associativity of processors' caches. The primary challenge in implementing LogTM's eager conflict detection is therefore to detect conflicts on data accessed in a transaction, but no longer in the cache.

In this section, I first describe a mechanism for tracking transactions' read and write sets. Next, I present two implementations of LogTM's eager conflict detection, LogTM-Dir and LogTM-Bcast, which use directory and broadcast coherence respectively. Both systems leverage the coherence mechanism for fast detection, but do not require the caching of transactional data.

4.2.1 Tracking Read and Write Sets with R/W Bits

Most transactions are expected to have small read and write sets and thus, operate entirely in the cache—i.e., all data accessed in a transaction remain cached until the transaction completes. In this case, a LogTM implementation can use cache meta-state, e.g., the cache coherence state to record the membership of each block in the current transaction's read or write set. The example systems that follow both extend data caches to store two additional bits with each cache line: the R bit, which denotes presence in the transaction's read set, and the W bit, which denotes presence in the transaction's write set.

4.2.2 LogTM-Directory

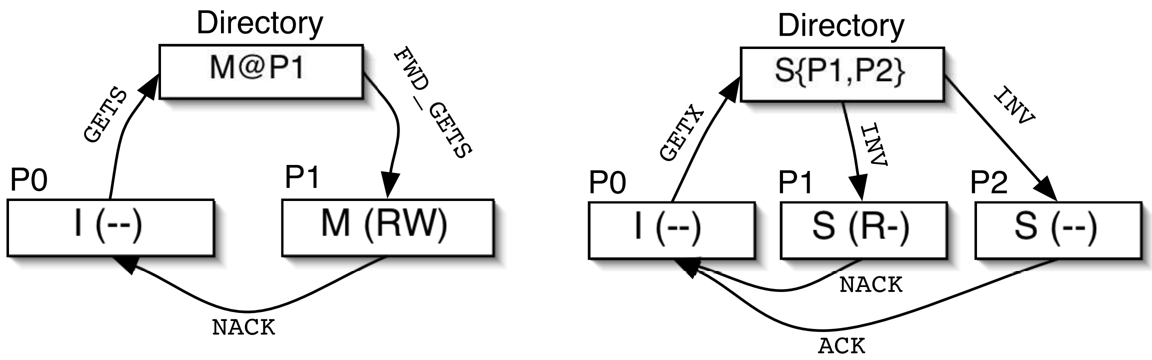
LogTM Directory (LogTM-Dir) extends a conventional MESI directory-based multiprocessor (MESI-Dir) with novel *sticky states* and support for *negative acknowledgements* (NACKs) to provide transactional conflict detection even when transactional data sets exceed the capacity or associativity of processors' private caches.

MESI-Dir. MESI-Dir is a cache-coherent non-uniform memory access (ccNUMA) multiprocessor loosely based on the SGI Origin multiprocessor [45]. Like the SGI Origin, MESI-Dir is comprised of several nodes, each of which contains: a processor (nodes in the SGI Origin had two processors), a region of memory and a directory memory. Also like the SGI Origin, MESI-Dir maintains coherence using a full-bit vector directory and an invalidation-based MESI coherence protocol. Both protocols support the clean exclusive state (E) with silent evictions—i.e., processors may evict lines in the E state without notifying the directory. Unlike in the Origin protocol, however, in MESI-Dir's protocol, the directory does not send speculative data replies for blocks owned by processors (blocks in states E or M). As a result, in MESI-Dir, a processor must send an extra message (the CLEAN message) to the directory in the case that it receives a forwarded shared or exclusive request for a block not present in its cache (e.g., a block that was formerly present in the E state and replaced silently from the cache). The directory responds to CLEAN messages by sending the data from memory to the original requester. The extra message causes MESI-Dir to be slower in this case, but omitting speculative data responses reduces the use of interconnection bandwidth.

LogTM-Dir. LogTM-Dir first extends MESI-Dir with NACK messages that signal transaction conflicts. Processors respond with a NACK to any coherence message that would otherwise require the violation of LogTM's strict two-phase requirement—i.e. a transaction conflict. When all transactional data are cached, LogTM-Dir behaves like the example implementation described in Section 3.2.2. Figure 4-3 illustrates two cases of transaction conflict: Figure 4-3 (a) depicts processor P0 attempting to read a block transactionally modified by processor P1; Figure 4-3 (b) shows P0 attempting to write a block transactionally read by P1. The blocks represent the state of a single memory block in each processor's cache and at the directory/ memory. The cache state includes both the coherence state (MESI) and the transactional R and W bits. In Figure 4-3 (a), the directory forwards P0's shared request (GETS) to P1. P1 checks its local cache for the requested block. Finding the block present and the W bit set, P1 responds with a NACK, alerting P0 of the conflict. In the second case, P0 sends a get-exclusive request (GETX) to the directory. The directory responds by sending invalidation messages (INV) to all of the processors on the sharers list (P1 and P2). Both P1 and P2 have shared copies of the block, but only P1 has read it as part of a transaction (i.e., the R bit is set in P1's cache). P2 responds with an ACK message according to the MESI-Dir protocol, but P1 seeing that the block is present in its cache and that the R bit for the block is set, responds with a NACK signalling the presence of a conflict to P0.

To detect transaction conflicts on data outside the executing processor's private cache, LogTM-Dir adds logical *sticky-S* and *sticky-M* states. These states allow the directory to forward all potentially conflicting memory requests to the executing processor, even after the block is evicted from the cache. The states are logical in that the behavior of the directory controller is the same for the S and sticky-S state and for the M and sticky-M states. The difference is that in a sticky state, the processor does not posses a valid copy of the data.

Because LogTM permits implementations to report false conflicts, LogTM-Dir could simply have processors respond with a NACK whenever the requested memory block is not present in the cache—i.e., when-



(a) (b)

FIGURE 4-3. In-Cache Conflict Detection in LogTM-Dir.

ever no R and W bits are available. Instead, to reduce the frequency of false conflicts, each processor maintains a single *overflow bit*, which is set when it evicts any block from its cache for which the R or W bit is set. When the overflow bit is set, a LogTM processor conservatively assumes that any invalidation messages or forwarded requests from the directory concerning blocks not present in its cache conflict with its current transaction and responds with a NACK. When the overflow bit is clear, however, LogTM-Dir responds positively according to the same protocol used by MESI-Dir (i.e., it responds by sending an ACK or CLEAN message).

A block enters the sticky-M state if a processor evicts a block for which the W bit is set. As shown in Figure 4-4 (a), the processor sends the modified data to the directory in a transactional write-back message (WB_XACT), indicating that the block is part of its current transaction. The directory controller receives the modified data from the evicting processor (the new version) and updates the in-memory copy of the block. But, because the block must still be isolated as part of a transaction, the directory does not change the coherence state of the block. Instead, the block enters the sticky-M state, wherein the directory refers all requests for the block to the evicting processor, which no longer possesses a copy of the block. Because the block was modified in the transaction, any request for the block (load or store) must result in a transaction conflict. When the evicting processor receives forwarded GETS or GETX requests from the directory for any block not present in its cache, it responds with a NACK.

A block enters the sticky-S state when a processor executing a transaction evicts a block for which the R bit is set (meaning the block was read as part of the current transaction). As in the base protocol, the eviction is silent and the evicting processor remains on the directory's sharers list. If the R bit is set, a LogTM-Dir processor sets its overflow bit. Because the block was read, but not modified in the transaction (otherwise the processor would have the block in the M state), only a store to that block will cause a conflict. Because the evicting processor is on the directory's sharers list, any store to the evicted block will cause the directory to send an invalidation message to the evicting processor. When it receives the invalidation message, the evicting processor first checks its local cache for the block. Seeing that the block is not present and thus precise information about the transactional status of that block (i.e., R/W bits) is not available, the evicting processor then checks its overflow bit to determine if the invalidation may indicate a possible transaction conflict. If the overflow bit is set, the processor NACK's the request to ensure safety.

A sticky-M state is usually cleaned on the first access to the sticky block following the termination (successful or otherwise) of the transaction that caused it to enter the sticky state. When a processor next issues a request for the block (by definition it cannot be valid in any processor's cache), the directory forwards the request to the block's former owner. Because the requested block is not present in its cache, the former owner responds to the forwarded request by checking its overflow bit. Assuming the overflow bit is clear, the former owner can rule out a possible conflict and respond positively to the request. The responding processor cannot supply the data, however, since it is not caching the block. This situation is identical to that which arises when a MESI-Dir processor silently replaces a block in the E state. As in that case, the processor sends a CLEAN message to the directory. Upon receiving this message, the directory knows that the data in memory is valid. The directory then responds by sending the data from memory to the requesting processor and changing the owner of the block to be the requesting processor. The directory changes the

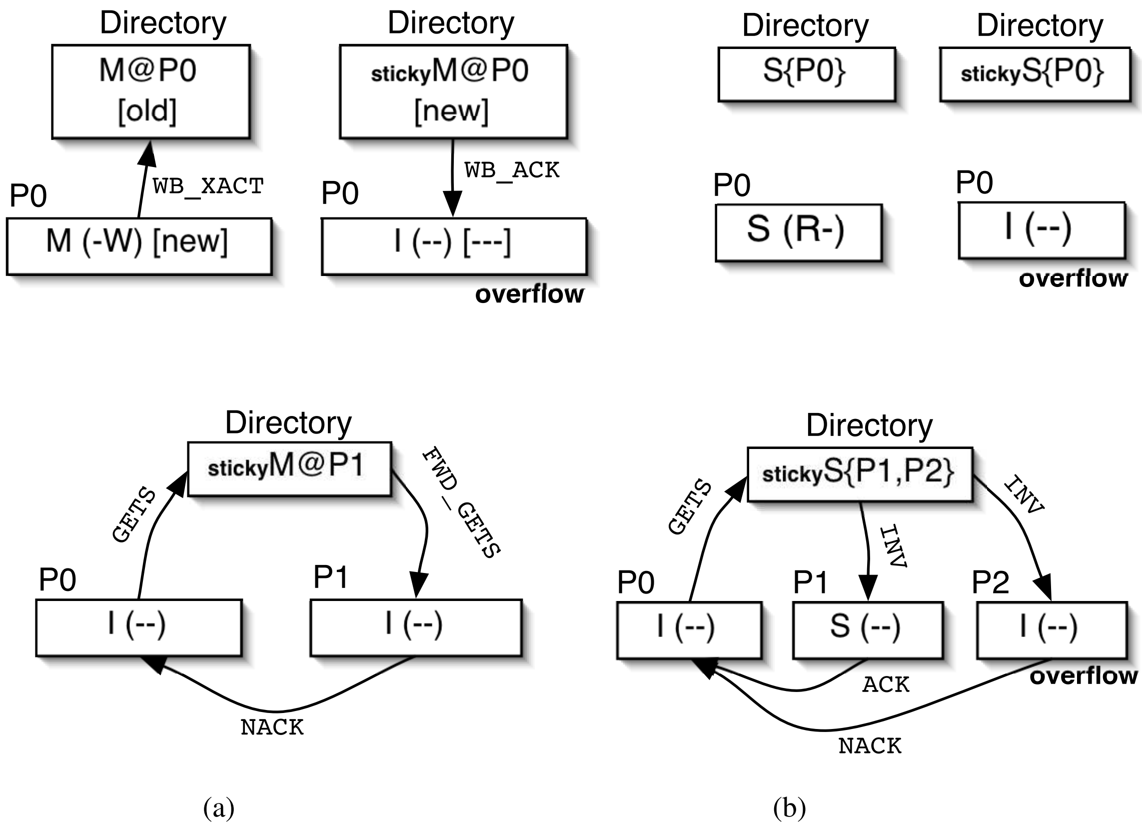


FIGURE 4-4. Conflict Detection on Un-Cached Data in LogTM-Dir Using Sticky States.

state of the block from sticky-M to E since the requestor will receive an exclusive copy of the data regardless of whether it issued a GETS or GETX request.

A sticky-M state will not be cleared, however, if the processor that is the former owner of the block is currently executing a subsequent overflowed transaction. In that case, the responding processor will not be able to distinguish whether the block in sticky-M was put in that state as a part of its current transaction or a previous one. Therefore, to ensure safety, it must conservatively NACK the request. Assuming that overflowed transactions are uncommon, such false conflicts are likely to be rare because they require not one, but at least two transactions to overflow on the same processor.

The rarity of overflowed transactions will prevent false conflicts from degrading performance in most situations, but the sticky states could potentially cause problems when more than one transactional application is running on the same machine. Specifically, if a thread from process A, which has recently run and committed a large transaction that overflowed the cache and left blocks in sticky states, is preempted and a thread from process B is scheduled in its place, then any overflowing transaction executed by the thread from B will potentially affect the performance of all threads in A. Although both applications will run correctly, the system will not be able to provide *performance isolation* between them [90].

4.2.3 LogTM-Broadcast

LogTM-Broadcast (LogTM-Bcast) is based on the AMD Hammer as described in [2, 50]. As according to the Hammer protocol, all requests in LogTM-Bcast are sent to memory, which, like a directory, either responds with data, or forwards the request to other processors. Unlike directory-based systems, however, memory does not maintain a list of sharers or a pointer to an owning node. Instead, all forwarded requests are broadcast to all nodes. The “directory” in an Hammer system simply serves as an ordering point for memory requests, but does not reduce coherence bandwidth.

For transactions in which all data remain in the executing processor’s private cache, LogTM-Bcast detects conflicts in essentially the same manner as LogTM-Dir. Cache lines are annotated with R and W bits to track the read and write set of the current transaction. Memory requests are sent to the directory, then broadcast to all nodes. Processors receiving GETS or GETX requests check the R and W bits for the corresponding line (if present in their cache). Each processor responds with an ACK or NACK depending on whether the R and W bits in their cache represent a conflict.

When transactional data overflow the cache, however, the behavior of LogTM-Bcast differs from that of LogTM-Dir. LogTM-Bcast does not need sticky states because all coherence requests are broadcast to all nodes. This guarantees that all potentially conflicting memory requests will automatically be sent to any

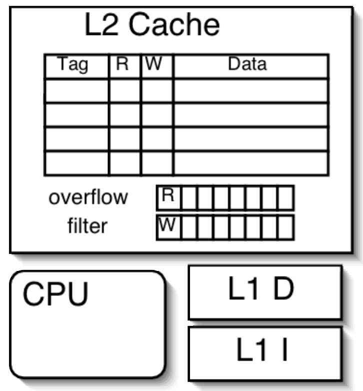


FIGURE 4-5. LogTM-Bcast Node.

processor executing a transaction, regardless of whether or not that processor has a valid copy of the requested block in its cache. Broadcasting, however, also means that many more non-conflicting requests will be sent to each processor. If LogTM-Bcast were to employ the single-bit overflow filtering scheme used in LogTM-Dir, every cache miss on every processor would generate a conflict with any transaction that overflows its processor's cache. Instead, LogTM-Bcast uses a Bloom filter [7] to store a conservative summary of the portion of the read and write sets that have overflowed the local cache.

Bloom filters encode membership in a set allowing *false positives*—reporting membership when the object is not present—but not *false negatives*—reporting absence when the object is present. Similar to a hash-table, an object's location in a Bloom filter is determined by a hash function. When an object is inserted into a Bloom filter, however, multiple indices are selected using multiple hash functions. The record at each location is then updated to indicate the presence of the newly inserted object (e.g., by setting a presence bit or incrementing a counter). To test for the presence of an object in the filter set, the locations corresponding to the result of each hash function are inspected. If any one of them is zero, the object is guaranteed not to be a member of the set.

Figure 4-5 shows a LogTM-Bcast node. Each node contains a processor, private L1 instruction and data caches, and a private unified L2 cache. Each node also contains read (R) and write (W) overflow filters.

During the execution of a large transaction, when a block is evicted from the cache that has been accessed in a transaction, its address is added to the filter. If the R bit of the block was set at the time of its eviction, its address is added to the R overflow filter. Both the R and W overflow filters are cleared along with the R W bits in the cache when a transaction completes. Because cache blocks are never removed from the read or write set of an active transaction, there is no need to store a count at each filter location. This makes the filters smaller and removes any concern over saturating the counters. The overflow filters are similar to the Bloom filters employed in JETTY [56] and transaction signatures proposed by Ceze et al. [11].

4.2.4 Discussion

In the event that a cache block containing a memory location in the read or write set of an active transaction is evicted from the executing processor's private cache, LogTM-Dir and LogTM-Bcast both continue to track accesses to the block by (1) ensuring that the executing processor is notified of (and has a chance to NACK) all conflicting memory operations and (2) filtering out many false conflicts at the processor. Which of these presents the greater challenge depends on the underlying coherence mechanism. In broadcast systems, like LogTM-Bcast, all potentially conflicting memory operations are trivially guaranteed to be sent to the executing processor because all memory operations that miss in a processor's cache are broadcast to all other processors. Because many non-conflicting memory accesses are sent to each processor, however, filtering out false conflicts becomes difficult. LogTM-Bcast includes an overflow Bloom filter to reduce the rate of false conflicts. In directory systems like LogTM-Dir, on the other hand, novel coherence extensions (e.g., sticky states) are required to ensure that conflicting memory operations are routed to processors that no longer contain the shared block. A directory, however, acts as a filter for coherence requests, which allows LogTM-Dir to filter out most false conflicts with a single overflow bit.

LogTM-Dir's directory tracks sharers with a full bit vector. Alternatively, some ccNUMA systems employ coarse-vector directories [28], in which each bit represents a group of processors. Coarse vector directories will increase the likelihood of false conflicts in LogTM. Because all processors in a group receive invalidation messages intended for any processor in the group, a coarse vector directory is a less effective filter of coherence requests. As a result, a system using a coarse vector directory will behave more like a broadcast protocol. To compensate, LogTM implementations that use coarse vectors could add an overflow filter like the one used by LogTM-Bcast. Since even a coarse vector director filters many coherence requests, such a system could presumably use smaller filters than LogTM-Bcast.

4.3 Implementing Conflict Resolution

LogTM-Dir and LogTM-Bcast implement conflict resolution (Section 3.3) using the *Requester Stalls* policy [9]. Both systems logically order transactions using the distributed timestamp method from Transactional Lock Removal [65]. To guarantee forward progress and reduce aborts, Requester Stalls only aborts a transaction that (a) could introduce deadlock and (b) is logically later than the transaction with which it conflicts. LogTM-Dir and LogTM-Bcast detect potential deadlock by recognizing the situation in which one transaction is both waiting for a logically earlier transaction and causing a logically earlier transaction to wait. This is implemented with a per-processor `possible_cycle` flag, which is set if a processor sends a NACK to a logically earlier transaction. Under the Requester Stalls policy, a processor aborts its transaction if it receives a NACK from a logically earlier transaction while its `possible_cycle` flag is set.

An additional challenge in implementing conflict resolution in LogTM is that conflicts are detected late in the processor pipeline. Most memory exceptions (e.g., TLB misses, page faults, access violations, etc.) occur early, before a load or store instruction triggers a request to the memory system. LogTM conflicts, however, are not detected until the memory system responds to memory instructions. This late notification is a potential problem for processors that maintain precise exceptions. By the time a LogTM implementation determines a load or store instruction has generated a conflict, subsequent instructions may be nearing completion or simply waiting to retire. The semantics of transactions, however, may reduce the importance

Predictor	State	Description
SINGLE-ENTRY	1 address per static transaction	Tracks 1 address per static location, updated on the first store of the transaction.
LOAD-PC	Map of load target address to load PC, list of upgrade load PCs	Tracks the PC of each load by target address in a transaction. If a load target is stored to, the associated PC is added to the upgrade list.
ALWAYS	None	Always requests exclusive access.

TABLE 4-1. Write Set Predictors

of precise exceptions for conflicts. If a LogTM processor executes a memory instruction that causes a transaction conflict, it may resolve that conflict by stalling that instruction, or aborting its transaction. In the first case, a late notification causes no harm. A stalled memory instruction behaves like a memory instruction waiting for a long-latency cache miss. In the second case, there is little need to capture the precise state of the current and subsequent instructions because the semantics of transactions dictate that their effects will be discarded.

4.3.1 Write Set Prediction

Because LogTM's eager version management makes aborts more costly, reducing the frequency of aborts can significantly improve performance. Transaction conflicts occur whenever the read or write set of one active transaction overlaps with the write set of another. Many of these conflicts can be resolved by stalling rather than aborting one of the transactions. Aborts are only necessary when a cycle forms between waiting processors. Frequently, this occurs when transactions read a memory location then write to that same location later in the transaction. This type of cycle can be broken by *write set prediction*—eagerly acquiring exclusive access on the first load to memory locations that will be modified later in the transaction. By acquiring write isolation early, the system detects the conflict while it is still possible to serialize the transactions.

Table 4-1 lists three simple write set predictors. SINGLE-ENTRY remembers the physical address of the memory location modified by the first store during the first execution of each static transaction. Whenever any transaction from the application subsequently loads that address, the transaction will eagerly acquire write isolation on that address. During transaction execution, LOAD-PC maintains a mapping of loaded addresses (load targets) to the program counter (PC) values of the corresponding load instructions. If any memory locations in the read set are later updated in the same transaction, LOAD-PC uses the load target-to-PC mapping to insert the PC of the load instruction into a predictor table. Subsequent executions of that load instruction trigger early acquisition of write isolation. ALWAYS acquires write isolation for all transactional loads and stores.

In their comparison of HTM systems, Bobba et al. found that the conflict resolution policy can have a significant impact on the performance of an HTM [9]. They also show that the Requester Stalls policy performs well in comparison to other schemes when combined with write set prediction.

4.4 Beginning and Ending Transactions

LogTM systems can implement the `begin_transaction` and `commit_transaction` calls in the LogTM API with new instructions *xbegin* and *xcommit*. The xbegin instruction first checks the TM Count register. If TM Count is zero, xbegin increments the TM Count register. If the system supports register checkpoints in hardware, xbegin takes a register checkpoint. The xcommit instruction decrements TM Count. If TM Count becomes zero, xcommit resets `Log Pointer` to `Log Base` and discards any register checkpoint.

Chapter 5

Evaluation

This chapter assesses the assumptions that underlie the LogTM system (Chapter 3) and presents an evaluation of the LogTM implementations presented in Chapter 4. Section 5.1 describes the methodology used in the evaluation, including system model assumptions. Section 5.2 describes the workloads used in this evaluation. Section 5.3 presents the overall performance of LogTM. Section 5.4 discusses the performance impact of various implementation trade-offs in LogTM.

5.1 Methods

The evaluation of LogTM presented in this chapter was performed using execution-driven full-system simulation of LogTM-Dir. To account for the variability in multi-threaded workloads, I introduce a small random perturbation in the memory response time as suggested by Alameldeen et al. [4] and run each simulation multiple times. The performance results shown below represent the average of these runs. The error bars shown are the 95% confidence intervals.

5.1.1 System Model

This chapter assumes the LogTM-Dir implementation described in Section 4.2.2. Table 5-1 summarizes memory system and processor parameters. The system includes up to 32 processors, each with two levels of private cache. A MESI directory protocol maintains coherence over a high-bandwidth switched interconnect. Though single-issue and in-order, the processor model includes an aggressive, single-cycle non-

Component	Settings
Processors	32, 1 GHz, single-issue, in-order, non-memory IPC=1
L1 Cache	16 kB 4-way split, 1-cycle latency
L2 Cache	4 MB 4-way unified, 12-cycle latency
Memory	4 GB 80-cycle latency
Directory	Full-bit vector sharers list; Directory cache, 6-cycle latency
Interconnection Network	Hierarchical switch topology, 14-cycle link latency

TABLE 5-1. System Model Parameters

memory IPC. A detailed model of the memory system includes most timing intricacies of the transactional memory extensions.

5.1.2 Simulation Platform

The simulation framework uses Simics [48] in conjunction with customized memory models built with the Wisconsin GEMS toolset [84]. Simics, a full-system functional simulator, accurately models the SPARC architecture but does not support transactional memory. The LogTM interface was instead simulated using Simics “magic” instructions—special no-ops that Simics catches and passes to the memory model. To implement the x*begin* instruction (Section 4.4), the memory simulator uses a Simics call to read the thread’s architectural registers and create a checkpoint. During a transaction, the memory simulator models the log updates. After an abort rolls back the log, the register checkpoint is written back to Simics, and the thread restarts the transaction.

The memory system simulator performs the rollback of transactional updates in simulated hardware and approximates the timing of software rollbacks. The execution time of a rollback is estimated using a fixed penalty to model the overhead of trapping to the abort handler plus a penalty for each entry in the transaction log to account for execution of the software handler itself. This abort penalty is applied in two ways. First, when a load or store instruction triggers an abort, execution on that processor is stalled for the abort penalty before the register state is restored and the transaction re-executed. Second, the W bits remain set

	Description	Settings	% Xact	Cycles/ Xact
B-Tree	Threads alternate between insert and lookup operations on a single tree. Each lookup or insert is performed in a transaction.	9-ary B-Tree, initially 5-levels deep. 20% update, 80% lookup	80.2%	1810
Shared Counter	All threads repeatedly increment a single counter.	2500 cycle average think time between transactions	3.0%	669

TABLE 5-2. Microbenchmarks

for the duration of the abort penalty, potentially stalling conflicting transactions. This approximation allows me to more easily measure the effect of abort latency on performance in LogTM (Section 5.4).

5.2 Workloads

LogTM's design is based on certain assumptions about the behavior of transactions. The wisdom of decisions such as favoring commits over aborts and providing support for large transactions will depend on the characteristics of transactions run on such systems. Due to the lack of software written for transactional memory, however, I can only make rough estimates of the behavior of transactions in future software. In this evaluation, I use two strategies to select applications on which to test LogTM: (1) using microbenchmarks that execute simple operations on common data structures and (2) converting critical sections in today's lock-based programs into LogTM transactions.

5.2.1 Microbenchmarks

The promise of transactional memory is to facilitate the writing of parallel programs. I demonstrate LogTM's ability to deliver this promise by measuring the scalability of the microbenchmarks described in Table 5-2. These simple programs were written with minimal regard for parallelization. They simply enclose high level operations on shared data in LogTM transactions. The first, BTree, contains two operations, lookup and insert. Lookup searches the tree for a particular key. Insert adds a new key-value pair to

Benchmark	Input	Synchronization Methods	% Xact	Cycles/Xact
Barnes	512 bodies	Locks on tree nodes	15.5%	5660
Cholesky	14	Task queue locks	0.456%	1800
BkDB	512 Operations	Locks	87.0%	81500
MP3D	4096 molecules	Locks	84.5%	2160
Radiosity	"largeroom"	Task queue & buffer locks	21.2%	4290
Raytrace	image=car	Work list & counter locks	68.4%	24600

TABLE 5-3. Benchmark Inputs and Characteristics

the tree, possibly splitting one or more nodes. Each operation is executed in its entirety in a single LogTM transaction. BTree creates a number of threads, each of which performs a random series of inserts and lookups. The second, Shared-Counter, creates a number of threads, each of which increments a single shared counter. The threads pause for a random think time between 0 and 5000 cycles. In addition to a brief description, Table 5-2 lists relevant inputs, the fraction of cycles spent executing transactions (% Xact) and the average length of each transaction in cycles (Cycles/Xact) for each microbenchmark.

5.2.2 Benchmarks

In addition to the microbenchmarks described above, this chapter evaluates LogTM on a set of benchmarks from the SPLASH [74] and SPLASH-2 [85] benchmark suites, and a benchmark based on the lock manager of an open source database. Table 5-3 describes the configuration of the benchmarks and the input parameters used to run them. The LogTM versions of these benchmarks replace locks with begin and end transaction calls. Barriers and other synchronization mechanisms were not changed. The lock versions use PARMACS library locks [6], which use test-and-test-and-set locks, but yield the processor after a predetermined number of attempts (one for these experiments). In one case (Raytrace), the benchmark code has been optimized for transactions by reorganizing a data structure to reduce false sharing. Reduced false sharing allows Raytrace to run much faster than the original program [54].

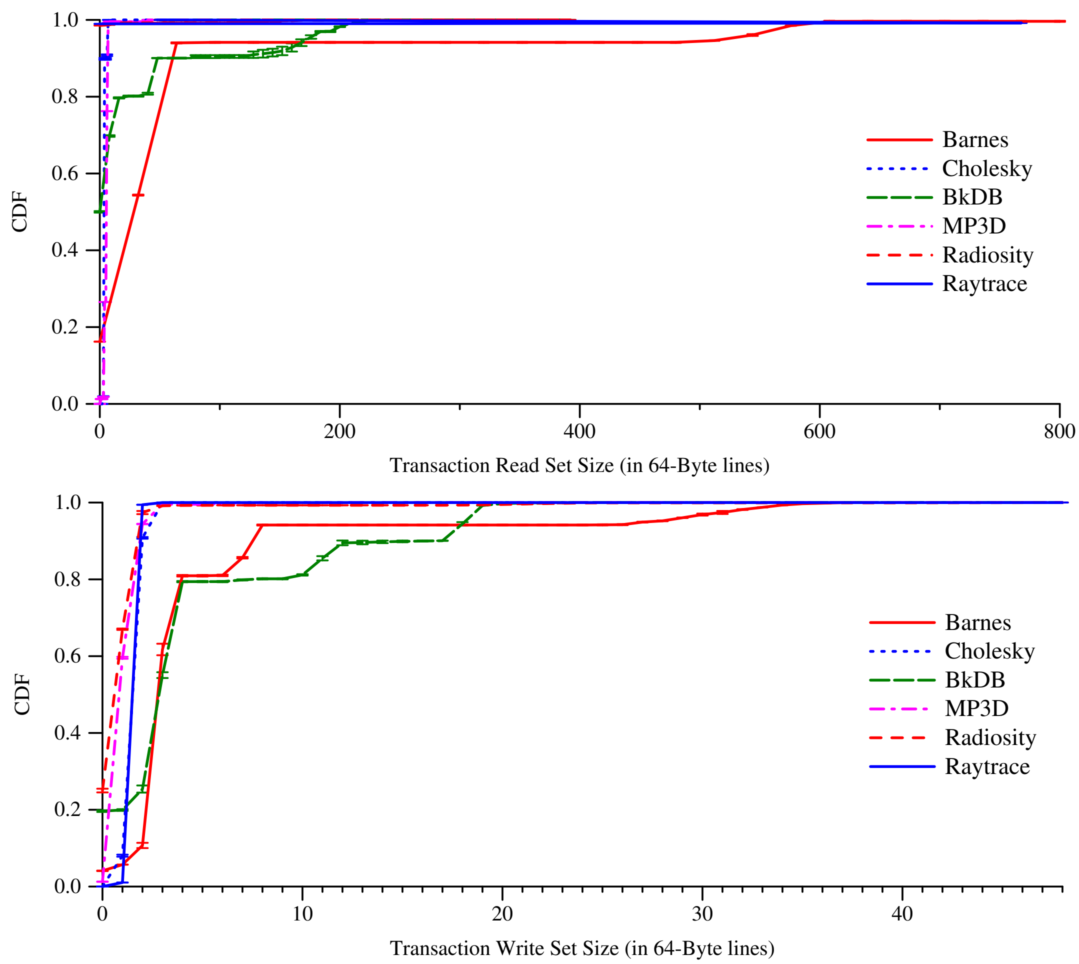


FIGURE 5-1. Cumulative Distribution of Transaction Read Set and Write Set Sizes.

Perhaps because they were adapted from lock-based programs, these benchmarks contain transactions that are generally short in duration and have small read and write sets. Table 5-3 includes the number of transactions and the average duration of transactions in each benchmark, and Figure 5-1 displays the distribution of the size of transaction read and write set sizes for each of the benchmarks measured in 64-byte blocks. Transactions in Radiosity, Cholesky and MP3D are short, and have small read and write sets. For the most part, Barnes and BkDB have the largest transactions. Barnes represents the simulated bodies in an octtree. The largest transactions re-balance this tree, writing approximately 30 memory blocks. BkDB uses transactions to coordinate updates to its lock table. Some transactions include the allocation of new lock objects, which requires writing to several queues and the new lock object. In Raytrace, most transactions are small, reading and writing just a single memory block. But, Raytrace also uses transactions to coordinate the allocation of memory from its free list. Some transactions traverse this free list, reading thousands of memory blocks.

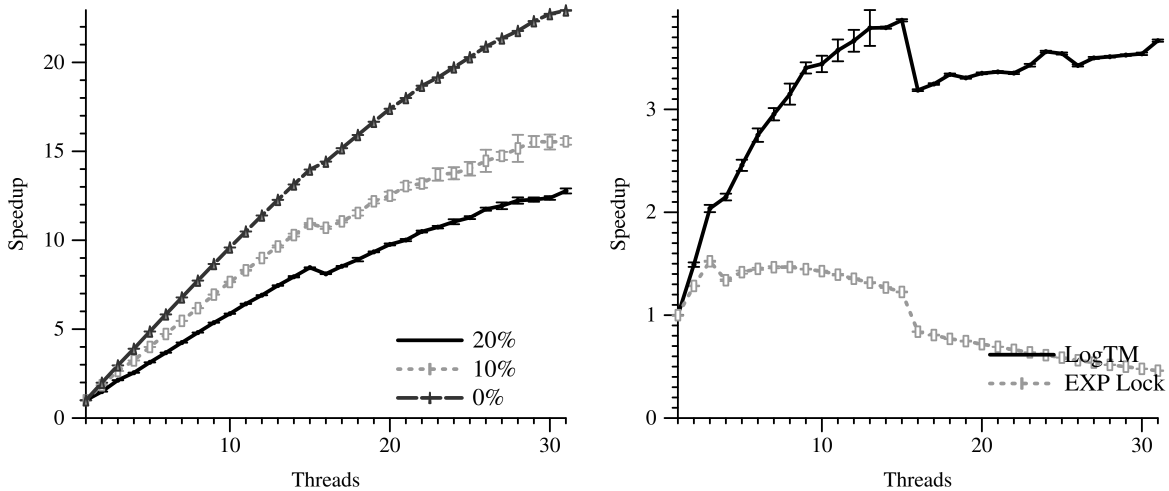
5.3 LogTM Performance

In order to demonstrate the potential of transactional memory in general and LogTM in particular, I present the scalability of the simple microbenchmarks described above. Next, I compare the relative scalability of the benchmark applications written using LogTM transactions to locks. Because the performance of LogTM relative to locks is dependent on many parameters, I choose one set of parameters for the overall comparison and analyze the effect of varying those parameters separately in detail.

The LogTM configuration in this experiment uses LOAD_PC write set prediction, buffered logging with a 64-entry log buffer, and assumes an abort handler trap latency of 200 cycles and a 40-cycle per-block overhead for restoring old values. Section 5.4 will evaluate the impact of these parameters in detail.

5.3.1 Microbenchmark Scalability

Figures 5-2 (a) and (b) display the speedup over single-threaded execution of the BTree and Shared-Counter microbenchmarks, respectively. Figure 5-2 (a) shows that the BTree benchmark scales well to 31 threads when the fraction of inserts is low. BTree reaches a speedup of 23 on 31 threads with no updates and a speedup of 10 with 10% updates. Although the speedup decreases to 8 with 20% updates, adding threads continues to be effective up to 31 threads. In the Shared-Counter benchmark, all threads contend for access to single counter and, in the lock version, its associated lock. Figure 5-2 shows that the LogTM transactions eliminate much of the overhead associated with acquiring and releasing the contended lock.



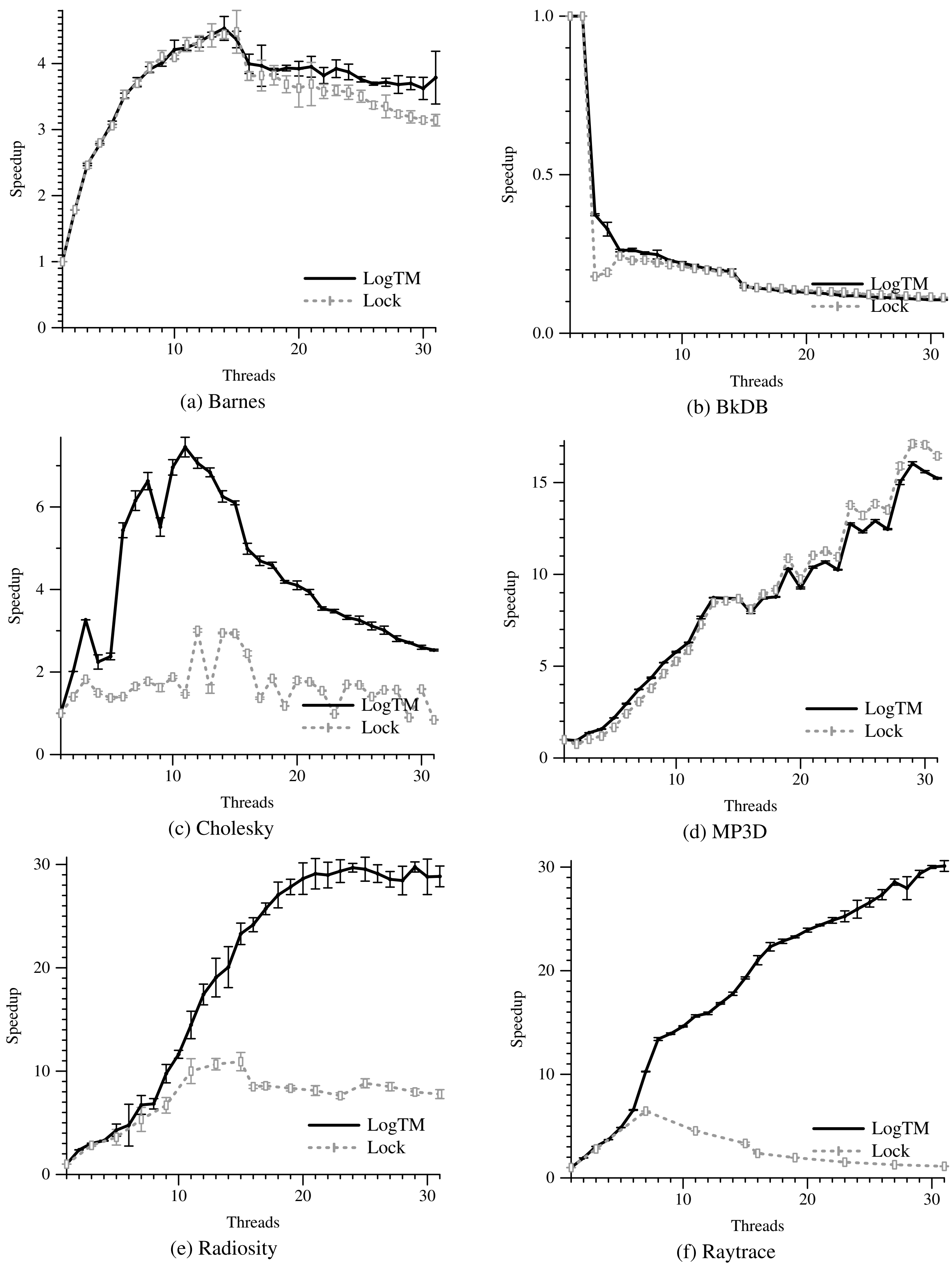
(a) BTree (b) Shared-Counter

FIGURE 5-2. Scalability of LogTM Microbenchmarks: (a) Scalability of BTree using LogTM Transactions with 0, 10 and 20% Updates, (b) Scalability of Lock-Based and Transactional Shared-Counter.

5.3.2 Benchmark Scalability

The results below demonstrate that LogTM improves scalability in several of the workloads described in 5-3, increasing the maximum achievable speedup for all benchmarks except Cholesky and BkDB, which do not scale with locks or transactions. In one case, Raytrace, LogTM also increases the number of processors used to achieve peak throughput.

Figures 5-3 (a)-(f) display the speedup of both lock-based and transaction-based benchmarks as the number of threads is increased from 1 to 31. For all of the benchmarks besides MP3D and BkDB, the LogTM version of the benchmark has a higher peak speedup than the lock-based version. For Barnes, the improvement is modest, but for Raytrace, Radiosity and MP3D, LogTM provides a dramatic increase in peak performance.



(a) Barnes

(b) BkDB

(c) Cholesky

(d) MP3D

(e) Radiosity

(f) Raytrace

FIGURE 5-3. Scalability of LogTM vs. Locks

For most of the benchmarks, the LogTM program and the lock program reach their peak performance at the same number of threads—Barnes at 15, MP3D at 29 and Cholesky at 12. Radiosity and Raytrace, on the other hand, scale much more effectively with LogTM, not reaching peak performance on LogTM until 22 and 31 threads, respectively, while lock-based Radiosity and Raytrace reach their peaks at only 15 and 3 threads.

5.4 Implementation Trade-offs

As discussed in Chapter 4, LogTM may be implemented in many ways, using varying levels of hardware support for log creation, log rollback, conflict resolution and write set prediction. In this section, I measure the performance impact of several of these trade-offs.

5.4.1 Write Set Prediction

One important factor in the performance of LogTM is the frequency of aborts. LogTM's eager version management is optimized for the case that aborts are rare and, as the results below demonstrate, its performance suffers when aborts are common.

I compare the three schemes for predicting write sets in transactions described in Section 4.3.1: SINGLE-ENTRY, LOAD-PC and ALWAYS. Figure 5-4 displays the relative abort rate—the ratio of aborts to all attempted transactions. An abort rate of zero means that all transactions commit, whereas an abort rate of one indicates that all transactions abort (livelock). SINGLE-ENTRY suffices to reduce the abort rate for several benchmarks, especially Raytrace, which frequently updates shared counters inside transactions. LOAD-PC and ALWAYS dramatically reduce the abort rate for all benchmarks. With LOAD_PC, only in BkDB do more than half of all transactions abort.

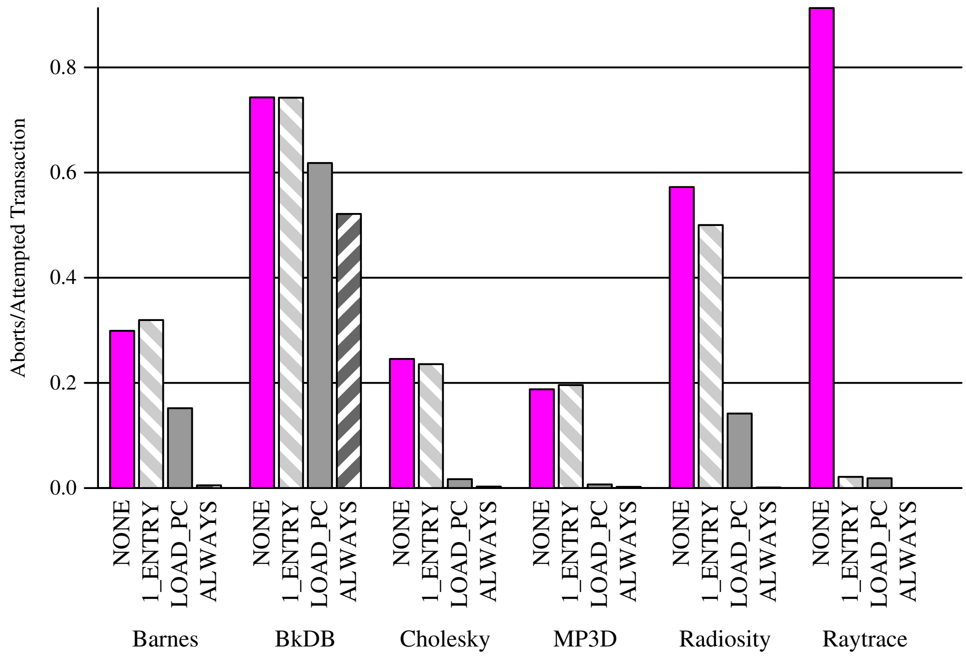


FIGURE 5-4. Transaction Abort Rates for Three Write Set Predictors.

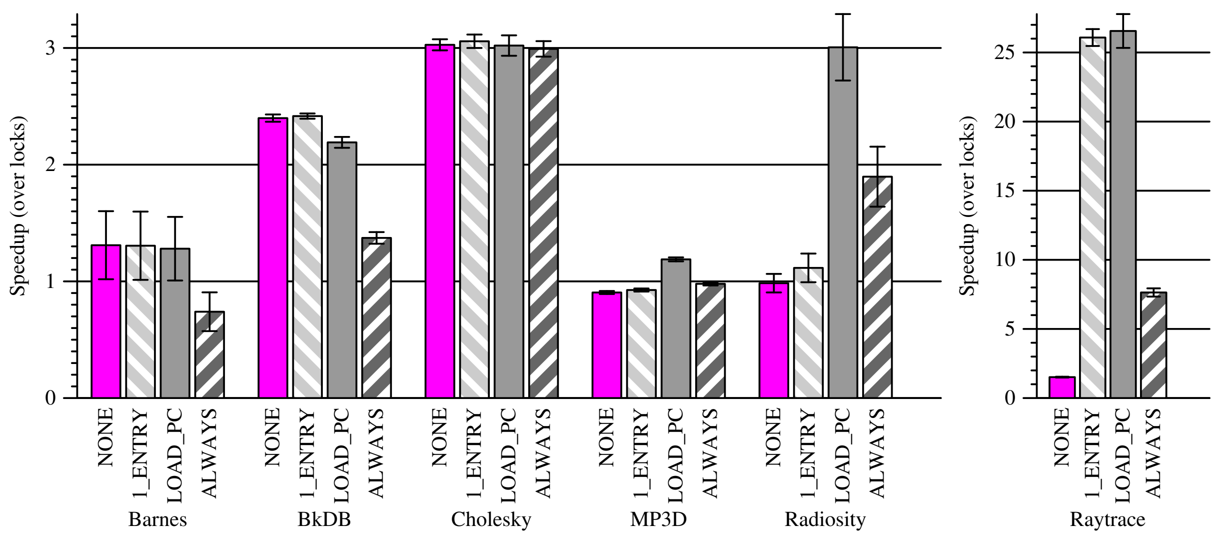


FIGURE 5-5. LogTM Performance with Write Set Prediction.

This reduction in abort rate, in most cases, leads to better performance in LogTM. Figure 5-5 displays the performance of LogTM using each of the write set predictors normalized to the performance of the lock-based program. As shown in Figure 5-5, write set prediction provides a substantial performance improve-

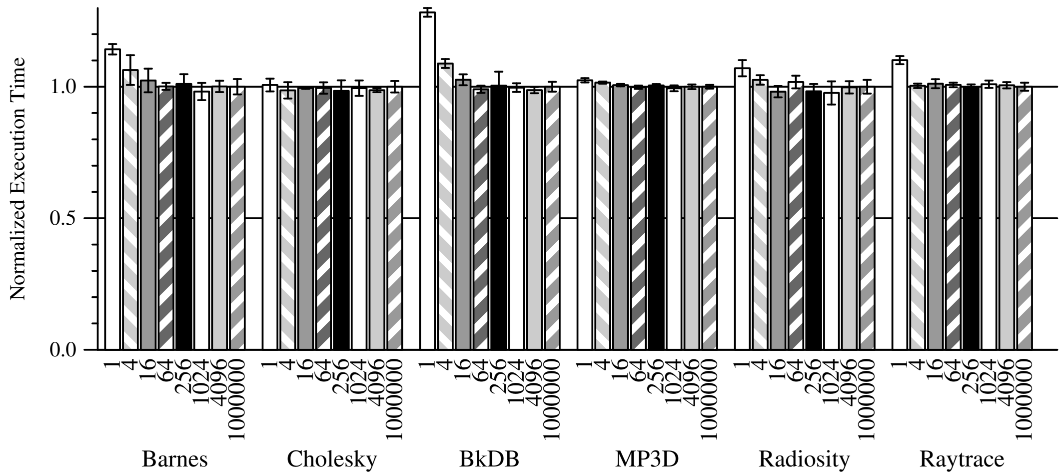


FIGURE 5-6. Performance Impact of Buffer-Spill Stalls.

ment in several workloads. In particular, write set prediction speeds up LogTM on the benchmarks on which it performs the worst. Overall, LOAD-PC provides the most consistent performance improvements. Although ALWAYS reduces aborts more than LOAD-PC, it also introduces more stalls, which erode the benefits of fewer aborts.

5.4.2 Hardware Support for Logging

Section 4.1.4 describes buffered logging. The log buffer reduces processor-to-cache memory traffic and eliminates the need to perform a virtual to physical address translation on the log pointer for each store. When the buffer fills, however, software spills the contents of the hardware buffer to the in-memory log. Figure 5-6 examines the effect of the size of the log buffer on LogTM performance. For several benchmarks, the size of the log buffer has little effect on performance at all. Only Radiosity and BkDB seem particularly sensitive to the size of the buffer. Even for those workloads, however, a reasonable size buffer (e.g., 64 entries) provides performance within 10% of that of an unlimited buffer.

Block Size	Barnes		BkDB		BTree		Cholesky		Mp3D		Radiosity		Raytrace	
	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk
4	19.6	3.92	4.97	3.98	22.4	4.00	2.91	4.00	4.80	4.00	4.26	4.00	2.02	4.00
8	12.7	6.04	4.39	4.50	18.2	4.92	2.77	4.20	2.79	6.88	2.69	6.33	1.98	4.08
16	8.68	8.86	3.88	5.10	12.0	7.47	2.64	4.41	2.03	9.47	2.13	8.00	1.98	4.08
32	6.80	11.3	3.80	5.20	7.51	11.9	2.59	4.49	1.62	11.8	1.90	8.95	1.98	4.08
64	5.51	14.0	3.32	5.96	4.76	18.8	2.00	5.88	1.47	13.1	1.75	9.72	1.98	4.08

TABLE 5-4. Log Size/Utilization at Varying Log Granularities

5.4.3 Log Granularity

Section 4.1 discusses the trade-offs involved in selecting the granularity at which to build the transaction log in LogTM. The optimal granularity depends on the size of transactions' write sets and the degree of spatial locality in them, both of which are dependent on the workload. For the workloads examined in this dissertation, the optimal logging granularity is quite small—likely 4 or 8-byte blocks.

Table 5-4 displays the average number of log entries per transaction (Blk/T) and average number of bytes used in each logged block (B/Blk) for logging at several different granularities. These results suggest that a small logging block size will be more efficient for these workloads. For many of the benchmarks, increasing the logging granularity (block size) does not result in a significant reduction of log entries per transaction. Of these benchmarks, only for Barnes and BTree does increasing the block size from 8 B to 64 B (an 8-fold increase) reduces the number of log entries by a factor of two or more. For Raytrace, increasing the block size beyond 4 B has almost no impact on the number of log entries.

Figure 5-7 compares the average size of the transaction log for logging performed at granularities from 4 bytes to 64 bytes assuming that addresses are 64-bits (8-bytes). Individual bars divide the log size into three components: Values (old values in the log), Address (virtual addresses of updated memory locations), and Unused (old values of memory locations not actually modified). For BkDB, Cholesky and Raytrace, the smallest logging granularity (4 bytes) results in the smallest log. These are the workloads with the least

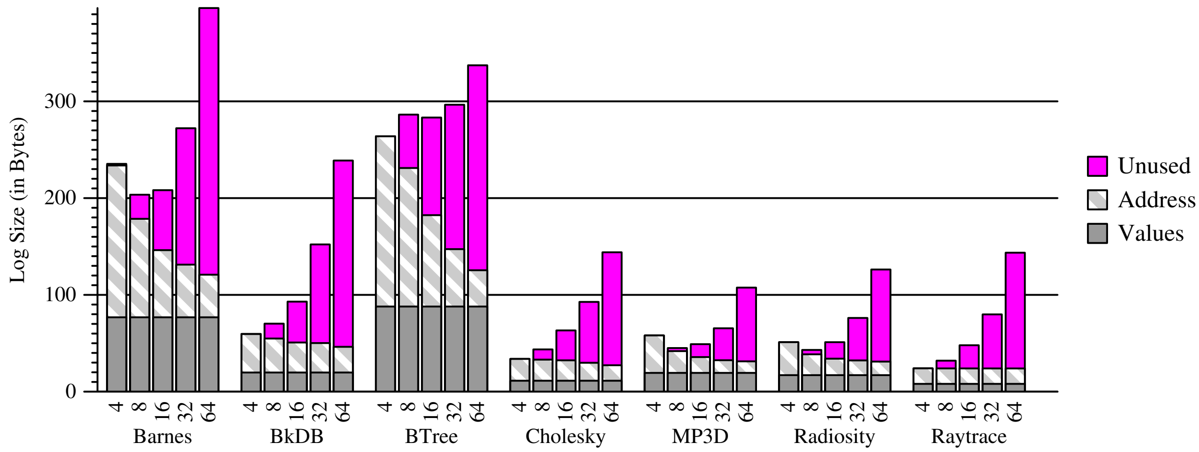


FIGURE 5-7. Effect of Logging Granularity on Log Size.

amount of spatial locality in transactions' write sets. In each case, increasing the logging granularity fails to reduce the Address component of the log. For Barnes, MP3D and Radiosity, an 8-byte logging granularity is most efficient. For these workloads, increasing the log granularity significantly reduces address overhead, but eventually also increases unnecessary logging. Up to 16-byte blocks, the increase in unused space is largely offset by the reduction in address overhead. Beyond 16-byte blocks, however, the unused space dominates other components, becoming more than half the log at a 64-byte granularity. Somewhat surprisingly, in BTree, logging large blocks reduces address overheads significantly, but logging 4-byte blocks is most efficient. Overall, logging at an 8-byte granularity seems to be the best fit for these workloads. Logging on 8-byte blocks is never much less efficient than logging at 4-byte blocks. Furthermore, most high performance computer systems now support 64-bit (8-byte) operations directly in hardware.

Interestingly, the benchmark for which a large block size results in the greatest reduction of log entries, BTree (22.4 to 4.76), was written using transactions from the outset rather than converted from a lock-based program. It may be the case that programs written specifically for transactional memory will use transactions that are longer-running and that write to more memory locations than the critical sections in programs written using lock-based mutual exclusion.

5.4.4 Abort Overhead

LogTM saves hardware complexity by implementing transaction rollback in software. To justify that strategy, I have argued that aborts are sufficiently rare that any overhead incurred in handling them will not affect performance. Figure 5-8 shows the effect of varying the overhead of handling aborts on the performance of LogTM-Dir. Recall that earlier experiments in this chapter have assumed a 200-cycle overhead to jump to the abort handler and a 40-cycle per memory block overhead to process undo records. The three bars in Figure 5-8 display the relative speedups of idealized abort processing, in which all aborts take only 1 cycle, the base 200 cycle/40 cycle implementation and a pessimistic model, in which trapping to the abort handler takes 1000 cycles and processing each undo record requires 200. Figure 5-8 (a) shows this comparison when the system uses the default LOAD_PC write set predictor. For the benchmarks where write set prediction is most effective (Raytrace, Radiosity, and MP3D), the abort overhead has almost no effect on performance. Of those benchmarks, only Radiosity is effected at all, and only slightly. Benchmarks for which write set prediction is not successful, however, suffer more significant performance degradations when abort overheads increase. Not surprisingly, large abort overheads affect BkDB and Barnes, which have the highest rate of transaction aborts, more than the other workloads.

One might expect that the sensitivity to abort delay is simply a function of the abort rate. Consider, however, the difference between Figure 5-8 (a) and Figure 5-8 (b). The same workloads are less sensitive to increased abort overheads when write set prediction is not used, despite the higher frequency of aborts. This counter intuitive behavior is a result of the difference in the way LogTM handles read-only and read-write data on abort. When a LogTM transaction aborts, it maintains write isolation on its write set during

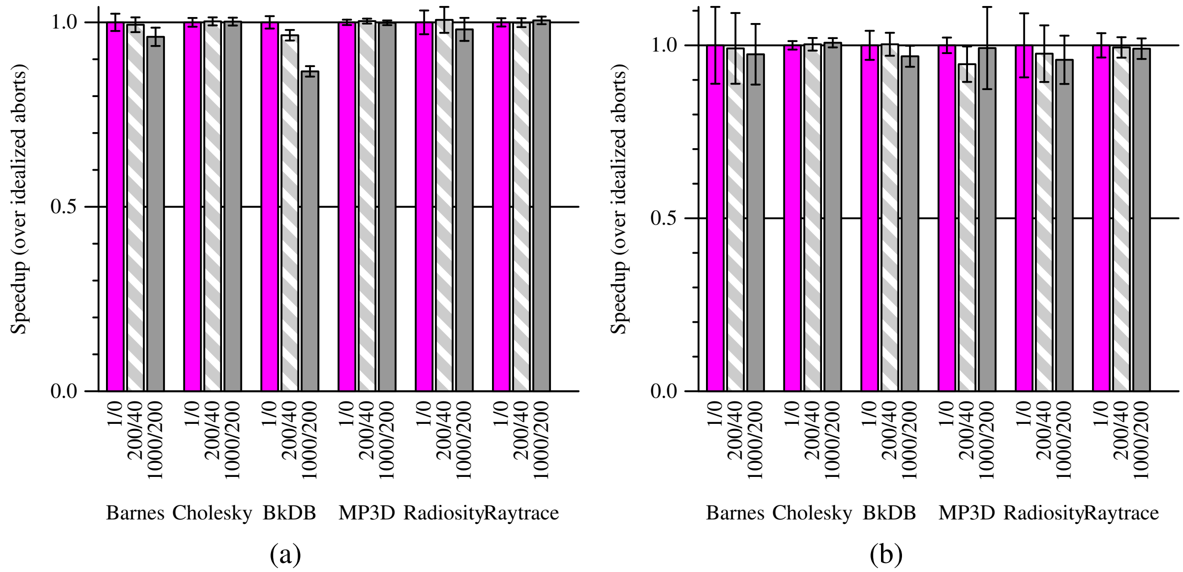


FIGURE 5-8. Affect of Abort Overhead on LogTM Execution Time: (a) With LOAD_PC Write Set Prediction, and (b) with No Write Set Prediction.

the execution of the abort handler. Doing so prevents other threads from observing updates from the aborting transactions. But, LogTM transactions release read isolation immediately because memory locations in the read set are not accessed during the abort. Write set prediction turns some reads into writes, expanding the effective write set of a transaction. By moving contended blocks from the read set to the write set, write set prediction, delays the transfer of those blocks from an aborting transaction to an active one, actually degrading performance when abort overheads are high.

5.5 Summary

In summary, the scalability of the two microbenchmarks, BTree and Shared Counter, which were written using obvious transactions, on LogTM demonstrates that transactional memory can simplify the writing of at least some parallel programs. Larger programs can also benefit from LogTM. The five benchmarks studied in this chapter all perform as well or better when using transactions than they do using locks; Raytrace and Radiosity perform much better. One of the keys to good performance in LogTM is to reduce the frequency of aborts. Stalling first to resolve conflicts combined with pc-based write set prediction reduces the frequency of aborts to less than 1 in 10 transactions for 6 of the 8 workloads studied and less than 2 in 10 for all workloads except BkDB.

For the most part, transactions in the workloads I studied are small, reading and writing only a few tens of bytes in the common case. Several of the workloads, however, have a few transactions that are much larger. Buffered logging is well suited to this distribution. Using reasonably sized buffers (e.g., 16 entries), most transactions' logs will fit entirely within the buffer. Since these workloads exhibit little spatial locality in the write sets of their transactions, logging at a small granularity, such as 8 bytes, reduces overall log size and could increase the efficiency of buffered logging.

Chapter 6

Extending LogTM

The implementations of LogTM presented in Chapter 4, and evaluated in Chapter 5 successfully abstract the size of hardware caches and buffers by supporting transactions whose read and write sets exceed the capacity of those structures. But, those implementations still restrict transactions in important ways. Since the introduction of LogTM in 2006 [54], researchers including myself have worked to extend LogTM to address these limitations. This chapter describes three distinct challenges and discusses how my collaborators and I have adapted LogTM to meet them. Our solutions demonstrate that despite the limitations of the original LogTM implementations, the LogTM framework can support fully-virtualized transactional memory with limited hardware support.

6.1 Nested Transactions

Software developers commonly create large applications by combining independently written modules. Ideally, such modules are *composable*—they may be combined in various ways and used with knowledge only of module interfaces, not internal mechanisms. But, writing composable modules that contain synchronization (particularly lock-based synchronization) has proven difficult [31]. For example, the lock-based move method in Figure 1-1 (a) cannot be safely included in a library module due to the risk of deadlock. In contrast, transactions compose naturally and transactional memory provides a means for programmers to write thread-safe modules that may be used without knowledge of internal mechanisms such as locking protocols.

Facilitating composability in transactional memory programs, however, requires that transactional memory systems support the *nesting* of transactions—starting and ending one transaction from inside another. Without such support, programmers would not be able to use libraries that contain transactions. When transactions are nested, we call the first transaction the *parent* and the transaction started from within the *child*. LogTM naively supports nesting by *flattening*, or subsuming child transactions into their parents. The LogTM systems presented in Chapter 4 implement flattening with a counter (TM Count), which is incremented at transaction begin and decremented at commit. Only transaction begins when the counter is zero and transaction commits that return the counter to zero are meaningful. In the database literature [22, 57, 83] and, recently, in transactional memory literature [51, 55, 58, 59, 60] researchers have developed two optimizations over flat nesting: (1) *closed* nesting with partial aborts and (2) *open* nesting, which have been shown to increase both the expressiveness of transactions and the performance of transactional systems.

6.1.1 Closed Nested Transactions (with Partial Abort)

In closed nesting, the read and write sets of a child transaction remain separate while the child is active, but merge with those of the parent when the child commits. This allows closed nested transactions to abort independently of their parent transactions. A closed nested transaction may access any updates made thus far by its (uncommitted) parent transaction without causing a transaction conflict. All of the child's updates, however, remain isolated from other threads until both the child and the parent have committed.

Closed nesting seeks to improve performance over flattening by aborting and re-executing only the child transaction (and not its parent) when possible. Consider the case of a long running parent transaction which rarely conflicts with other transactions or aborts, but which calls library function that uses a transaction to access a highly contended data structure. With flat transactions, conflicts in the child transaction can cause the entire long-running parent transaction to abort. Closed nesting eliminates such costly and unnecessary rollbacks.

Moravan et al. develop extensions to LogTM, specifically LogTM-Dir, to support closed nesting [55]. The solution they propose, called Nested LogTM, involves primarily: (1) segmenting the transaction log (Section 3.1.1) into frames—one for each nested transaction—and, (2) replicating conflict detection state, e.g., R/W bits (Section 4.2) for each level of nesting supported.

Beginning a nested transaction in Nested LogTM is similar to beginning a top-level transaction in LogTM-Dir. The system allocates a new frame in the transaction log and logically checkpoints the register state. The hardware also allocates a new set of R/W bits in the cache. Subsequent loads and stores update these bits leaving the parents' read and write set unchanged. If the nesting depth of a transaction exceeds the number of sets of R/W bits, Nested LogTM resorts to flattening transactions.

Conflict detection in Nested LogTM is identical to its counterpart in flat LogTM except that each coherence request checks all sets of R/W bits. If a conflict is found at any nesting depth, the request is NACK'ed. Additionally, the coherence hardware tracks the shallowest nesting depth that may be involved in a cycle. That depth is then passed to the abort handler to ensure that an abort unrolls only as many transaction levels as is necessary to resolve the pending conflict.

Committing a nested transaction requires two additional steps. First, the committing child transaction's log frame is merged with that of its parent by setting a committed flag in its header. The header then becomes a *garbage header*, which occupies space in the log, but is ignored on abort. Second, the R/W bits of the committing child are merged with those of its parent. This merge can be implemented efficiently with a 'flash-OR' circuit [55].

Supporting the partial aborts themselves is simple in LogTM since aborts are implemented in software. Nested transactions are rolled back independently by unrolling the corresponding frames in the transaction log. If more than one level of nesting is aborted, the abort handler unrolls frames in LIFO order, just as it reads the individual undo records in each frame in LIFO order.

6.1.2 Open Nested Transactions

Open nested transactions seek to increase concurrency and to provider richer semantics for transactional programming. Open nesting relaxes the atomicity and isolation guarantees of closed transactions by committing child transactions independently of their parent transactions. Specifically, when an open child commits, it releases isolation on all memory locations it has accessed, allowing other threads to see its effects before its parent has committed. This allows long-running transactions to access contended resources without overly restricting concurrency. For example, a long-running transaction that allocates memory without open nesting must maintain isolation on the free list until it commits. A long running transaction that allocates memory in an open nested transaction only needs to hold isolation on the free list for the duration of the allocation.

Open nested transactions behave like closed nested transactions until commit. Whereas in close nesting the read and write sets of the child are merged with that of its parent, in open nesting, when the child transaction commits, it's read and write sets are cleared and its updates are exposed to all threads.

Open nesting's relaxation of atomicity and isolation increases concurrency, but adds significant complexity to the programming model. One source of this complexity is the fact that open nested transactions are not atomic with their parent. If a parent transaction aborts after it has executed an open nested child, the child transaction, which has committed is not rolled back. Instead, open nested transactions may register *compensating actions* to perform a logical reversal of the operation performed in the open nest if its parent aborts. Designing compensating actions requires knowledge of the semantics of the program and is therefore left to programmers.

Moravan et al. extend LogTM to support open nesting by adding compensating action records to the log and clearing R/W bits on commit. Open nested transactions begin exactly like their closed counterparts—the system increments the nesting level and allocates a new log frame, saving the register state. Increasing the nesting level, as in closed nested transactions, allocates a new set of R/W bits for the child transaction. Open commits, however, are quite different. The commit of an open nested transaction clears the child's conflict detection state (e.g., R/W bits) and log frame instead of merging them with that of the parent. Isolation is released on blocks accessed by the committing child transaction by flash clearing its set of R/W bits. The child's log frame is cleared by resetting the log frame pointer to parent's header and the log pointer to the child's header (allowing it to be overwritten). After clearing the log and R/W bits, the system writes a compensating action record to the log. The compensating action record specifies a function to be called and a list of arguments to be passed in case the parent aborts. If the parent aborts, the software abort handler replays the log in reverse order alternately restoring values when it encounters undo records, and performing compensating actions when it encounters compensating action records.

6.2 Virtualizing Conflict Detection

LogTM's version management is well suited to virtualization. Storing the transaction log in virtual memory means that log values may be evicted from the cache and even paged to disk without affecting the state of a transaction. Furthermore, using virtual addresses in the transaction log means that pages accessed in a transaction can be re-mapped to different physical addresses without affecting transaction version management. Virtualizing conflict detection, however, is more challenging.

Conflict detection is more difficult to virtualize in all HTM systems because, unlike version management, conflict detection must be performed even for suspended transactions. Systems that employ eager conflict detection must check all loads and stores against the read and write sets of all active transactions, including those that are suspended. Systems that employ lazy conflict detection must validate committing transactions against the updates of all transactions that ran concurrently, including those that ran when the transaction was suspended.

Virtualizing conflict detection is particularly challenging in LogTM, because LogTM requires hardware conflict detection for transactions of any size. Recall that both LogTM-Dir and LogTM-Bcast implement conflict detection using read (R) and write (W) bits for blocks in processors' caches and sticky states and bloom filters respectively for evicted blocks (Section 4.2). These mechanisms present a challenge to virtualization for two reasons: (1) their state is not easily saved and restored and (2) they map read and write sets based on processors and physical addresses rather than threads and virtual addresses.

6.2.1 Detecting Conflicts with Signatures

Yen et al. [87] address the difficulty of saving and restoring R/W bits in LogTM by instead tracking read and write sets using *signatures* [11]. Signatures are compact encodings of sets of addresses. More precisely, a signature representing a set of addresses, S, encodes a superset of S because many sets of addresses may alias to the same filter value.

A signature implements several operations. Let O be a read or a write and A be a block-aligned physical address. INSERT(O, A) adds A to the signature's O-set. Every load instruction invokes INSERT(read,A) and every store invokes INSERT(write, A). CONFLICT(read, A) returns whether A may be in a signature's write set (thereby conflicting with a read to A). CONFLICT(write, A) returns whether A may be in a signature's read- or write-sets. Both tests may return false positives (report a conflict when none existed), but may not have false negatives (fail to report a conflict). Finally, CLEAR(O) clears a signature's O-set.

Signatures may be implemented in a variety of ways including using a Bloom Filter [7]. Ceze et al. [11] and Yen et al. [87] describe several signature implementations.

Signature aliasing can cause some non-conflicting memory requests to be incorrectly interpreted as conflicts. Such *false conflicts* serialize the execution of transactions. Serialized execution, however, is legal in the transactional execution model. Signature aliasing thus affects performance, but not correctness. As a result, signatures are amenable to LogTM's conservative conflict detection, where the system must detect all conflicts, but may report false conflicts.

Most importantly, signatures can be saved and restored by software. Yen et al. take advantage of that capability to virtualize conflict detection in LogTM, calling their new design LogTM-Signature Edition, or LogTM-SE. Unlike the implementations of LogTM presented in Section 4.2, LogTM-SE, tracks transactions' read and write sets with a read and write signature for each processor. LogTM-SE also maintains a *summary signature*, which encodes the aggregate read and write set of suspended transactions. The ability to save and restore the active signature and to track the state of suspended transactions in the summary signature allows LogTM-SE to supporting unbounded nesting, thread suspension and migration and paging.

6.2.2 Supporting Unbounded Nesting with Signatures

Nested LogTM [55] uses the transaction log to provide version management for an arbitrary number of nested transactions, but can only support conflict detection for a fixed number of levels. LogTM-SE overcomes that limitation by tracking the aggregate read and write set of the parent and child transaction together in hardware and using the flexible storage in the transaction log to save conflict detection state for each nested transaction.

LogTM-SE's nested conflict detection is based on the observation that clearing the read and write set of a child transaction is equivalent to restoring the read and write set—or read and write *signature*—that held when the child transaction began. Therefore, LogTM-SE can independently clear the read and write sets of committing or aborting child transactions by restoring a checkpoint of the signature state when before the child began. Table 6-1 lists the effect of beginning, committing and aborting nested transactions on the

Operation	Logical Effect	LogTM-SE Action
Begin Nested Transaction	no effect	Save read and write signatures to log.
Abort Nested Transaction	clear child read and write set, parent read and write sets unchanged	Restore signature checkpoint
Closed Nested Commit	merge parent and child's read and write sets	Mark child header as 'garbage' (discards signature checkpoint)
Open Nested Commit	discard child's read and write sets	Restore signature checkpoint

TABLE 6-1. Nested Conflict Detection in LogTM-SE.

read and write sets of active transactions and the actions taken by LogTM-SE to track that state. LogTM-SE saves the current hardware signature (read and write) to the log on each nested transaction begin. On commit of a closed transaction, no action is necessary because the hardware signature already represents the combination of the parent and child's read and write sets. To abort a transaction (or commit an open transaction), LogTM-SE simply restores the signature saved on the log.

6.2.3 Thread Switching and Migration

In LogTM-SE, all of a thread's transactional state—its version management and conflict detection state—is accessible to the operating system (OS). The version management state—old and new versions of transactional data—reside in virtual memory and do not need to be saved, moved, or updated on thread switches. A thread's conflict detection state can be saved by copying the read/write signatures to the log's current header. However, the hardware must continue to track conflicts with the suspended thread's signatures to prevent other threads from accessing uncommitted data. For example, another thread in the same process may begin a transaction on the same thread context and try to access a block in its local cache. The system must check this access to ensure that the block is not in the write-set of a descheduled transaction.

The challenge is to ensure that all active threads check the signatures of descheduled threads in their process on every memory reference.

LogTM-SE achieves this goal using an additional hardware signature, the summary signature, which represents the union of the read and write-sets of all suspended transactions. The OS maintains the following invariant for each active/summary signature pair: If thread T of process P is scheduled to use an active signature, the corresponding summary signature holds the union of the saved signatures from all descheduled threads from its process P. On every memory reference, including hits in the local cache (both transactional and non-transactional), LogTM-SE checks the summary signature to ensure that the request does not conflict with a descheduled transaction. Multi-threaded cores, where each thread on a core may belong to a separate process, require a summary signature per thread context.

The OS maintains, in software, a summary signature for the entire process and coordinates the updating of the hardware summary signatures on each processor. When descheduling a thread, the OS merges the thread's saved signatures into its process summary signature. It then interrupts all other thread contexts running threads from the process and installs the new summary signature. Any memory request that conflicts with a saved signature immediately traps to a conflict handler, since stalling is not sufficient to resolve a conflict with a descheduled thread. When the OS reschedules a thread, it copies the thread's saved signatures from its log into the hardware read/write signatures. However, the summary signature is not recomputed until the thread commits its transaction, to ensure that blocks in sticky states remain isolated after thread migration. The thread executes with a summary signature that does not include its own signatures, to prevent conflicts with its own read- and write-sets. On transaction commit, LogTM-SE traps to the OS, which pushes an updated summary signature to active threads. Thus, with a single additional hardware signature per thread and small changes to the operating system, LogTM-SE supports both context switching and thread migration.

The cost of context switching within a transaction is relatively high, and for that reason we expect operating systems to support preemption control mechanisms [80] that defer context switches occurring within a transaction if possible. In addition, aborting short transactions may be preferable to incurring the overhead of propagating new summary signatures.

6.3 Software Contention Management

The ability to suspend active transactions LogTM-SE provides, however, poses a potential performance problem for LogTM. The conflict resolution policy used by LogTM-Dir (the Requester Stalls policy) will perform poorly if an active transaction conflicts with a suspended transaction. In the Requester Stalls policy, transactions only abort when they are both stalling and stalled-by an older transaction. Therefore, a suspended transaction (if is not stalled) might continue to stall an active transaction and not abort until it resumes execution. Scherer and Scott have developed several contention management policies for STM that consider suspended transactions [71].

LogTM can be extended to support software contention management by trapping to software on transaction conflicts. Load and store instructions that trigger aborts in LogTM already transfer control to a software handler. Implementing software contention management would simply entail extending that mechanism to be called on all conflicts rather than only on aborts. The `register_abort_handler` method in the LogTM API (Section 3.4) would be replaced by a `register_contention_manager` method. Applications would not need to register an abort handler in such a system because the abort handler could be called by the contention manager.

6.4 Summary

Each of the extensions described above follows a similar pattern. Nested LogTM segments the transaction log, exposing the version management state of nested transactions to software. The software handler uses that state to unroll nested transactions. LogTM-SE exposes conflict detection state to software, allowing the OS to suspend, resume and migrate threads without aborting active transactions. To support software contention management, LogTM exposes the presence of conflicts to software, which can then implement a host of policies. In each case, the behavior of the transactional memory system is made more flexible by granting software access to more hardware state and more control over system policies. None of these extensions alters the fundamental LogTM strategy: perform conflict detection conservatively (but quickly) in hardware, and version management (mostly) in software by updating memory in place and logging old values. Although they may supplant earlier LogTM implementations, these extensions attest to the flexibility of the LogTM framework for transactional memory.

Chapter 7

Related Work

LogTM builds on and has been influenced by the substantial body of research in transactional memory and other areas. This chapter outlines some of the most pertinent related work including other transactional memory proposals. Section 7.1 outlines other HTM schemes and contrasts them to LogTM. Section 7.2 discusses STM systems similar to LogTM. Section 7.3 describes various software transactional memory and hybrid hardware/software transactional memory schemes. Section 7.4 discusses the similarities and differences between transactional memory and thread-level speculation.

7.1 Hardware Transactional Memory

LogTM is one of several HTM systems, each of which uses a different techniques for version management conflict detection, especially when transactions are long running or touch large amounts of memory. Here, I revisit alternative proposals and contrast them to LogTM.

7.1.1 Unbounded Transactional Memory

Of the various transactional memory systems, LogTM is most similar to the work of Annanian et al. [5] who propose Unbounded Transactional Memory (UTM) to support transactions without any limitation on size or run time and Large Transactional Memory (LTM) to support merely large transactions with less hardware overhead. UTM is an idealized memory system designed to support atomic memory transaction of any length and any footprint (limited only by the size of virtual memory). Threads running a UTM transaction may be suspended or even migrated to other processors without aborting the transaction. UTM transactions are not required to abort if they access data that is not currently in physical memory nor if data they have modified is paged to disk.

Unlike many transactional memory systems, including LogTM, UTM does not rely on the cache coherence mechanism to detect transactions conflicts. UTM maintains transactional atomicity and isolation with a transaction log that is both a before-image log of transactional updates (like LogTM's log) and a cache block-granularity lock table, which it uses to enforce a strict two-phase locking concurrency control policy. Each running transaction has a separate log consisting of a commit record, which stores the status of the transaction (pending, committed or aborted), and a series of log entries. Each log entry contains: (1) the address of the memory block to which the entry applies, (2) the pre-transaction value of that memory block and (3) a pointer to the last log entry for the block, if one exists. These pointers form a linked list of readers, which contains the identities of all the transactions that have read the block. In addition to the log, UTM stores extra state for each block in memory, which consists of a pointer to the most recent log entry for this block and a bit, which indicates whether the block has been modified or merely read as part of a transaction. Together with the log pointer (null indicates no transaction), the bit encodes a shared or exclusive lock on the corresponding memory block.

Compared to UTM, LogTM requires less hardware support. Unlike LogTM, UTM adds state to each block in memory. UTM uses this extra state to store pointers to its log, which it uses both version management and conflict detection. Furthermore, UTM manages this extra state completely in hardware, whereas LogTM delegates complex actions such as aborting transactions and managing transaction contention to software.

Like LogTM, Large Transactional Memory (LTM) [5, 46] uses the coherence protocol to detect conflicts for cached data. A single additional bit in the cache tags tracks the cache lines accessed in the current transaction. The cache also aids version management. The cache stores the new value of memory locations modified in the transaction, while main memory retains the pre-transaction value. The requirement that memory keep the pre-transaction value of modified memory locations requires the cache to write back the pre-transaction value of a modified memory location before it is overwritten in a transaction. For example, if a processor modifies the same cache line in multiple transactions, the cache must write back the line to memory in each transaction.

LTM supports transactions whose footprints exceed the capacity of the cache by spilling transaction state to a reserved (un-cached) region of memory. When a long transaction exceeds the capacity of a cache set, the hardware sets a per-set "overflow" bit and writes the overflowing transactional data to a hashtable in un-cached memory. The hash table is established by operating system software, but directly accessed and updated by the hardware. Subsequent loads to an overflowed cache set must also check the hashtable (which functions effectively as an in-memory victim cache). The cache controller must also search the hash table before responding to any coherence request that indexes to an overflowed cache set. While the controller is searching the hash table, all incoming cache interventions are stalled using NACKs.

LTM differs from LogTM in both its version management and in the way it divides the work of maintaining transactional memory semantics between hardware and software. When transactions fit in the cache, LTM stores new values in the cache and relies on memory to hold old values. This policy forces LTM to write dirty blocks back to memory before they are modified in a transaction and requires special action when transactional blocks are evicted from the cache. In contrast, LogTM always stores old values in separate memory locations—the log—allowing both new and old values to be evicted independently without any additional actions. Like UTM, LTM implements transactional memory completely in hardware.

7.1.2 Transactional Memory Coherence and Consistency

Transactional Memory Coherence and Consistency (TCC) [29] seeks both to implement atomic memory transactions and to simplify the design of multiprocessor memory systems by executing all programs as a series of atomic transactions. TCC is specifically designed for CMPs. It takes advantage of the high bandwidth communication that is possible between processors on the same chip. TCC is based on optimistic concurrency. Each processor executes transactions speculatively. Processors keep an extra bit of state per word in the cache to track the read set of each transaction. At the end of a transaction, processors broadcast their updates to all other processors. Processors receiving commit messages check each address against their read set. If there is a conflict, the speculative transaction is aborted and restarted. TCC relies on keeping transactional data in the private local cache and victim buffer of each processor in most cases. TCC can maintain atomic semantics when the data set of the transaction exceeds the on-chip buffers, but only by serializing transactions. Specifically, the processor executing the long-running transaction must obtain and hold the commit privilege until the transaction completes. This restricts the other processors to only performing speculative operations [29].

TCC differs from LogTM in both version management and conflict detection. Unlike LogTM, TCC uses lazy version management—storing new values in a per-thread write buffer. Also unlike LogTM, TCC employs lazy conflict detection—invalidating conflicting transactions on each commit. LogTM also differs from TCC in its ability to continue to execute transactions in parallel after a cache eviction.

7.2 Software Transactional Memory

The emergence of CMPs has also spurred research in transactional memory systems implemented completely in software, or software transactional memory (STM). These systems uses conventional synchronization techniques (blocking and non-blocking) in low-level libraries or in compiler generated code snippets to implement application-level transactions on today's hardware. Like HTMs, STMs vary in their implementation of version management and conflict detection. STMs also differ in the granularity on which each operation is performed. Thus far, most STMs perform conflict detection and version management on a block or object granularity. Many object-based STMs use lazy version management and non-blocking conflict detection and resolution [30, 32, 49]. These systems use an extra level of indirection (e.g., through a locator object) and atomic read and update operations to atomically switch versions on transaction commit.

7.2.1 Lock-Based Transactional Memory

Of all the STM systems developed so far, LogTM is most similar to lock-based STMs, such as the Multi-Core Runtime System (McRT) developed by Saha et al. [69], and Ennals' blocking STM [20]. Like LogTM, McRT implements eager version management, allowing transactions to update memory in place and saving old values to an undo log. McRT also uses a strict two-phase locking algorithm for protecting memory locations written in a transaction. Unlike LogTM, however, McRT uses read-versioning for read-only data instead of read locking. In an STM, read locking is far more expensive than in an HTM. To acquire a read lock, a thread must perform an atomic update of the lock word, which will invalidate that word in all other processors' caches and may disrupt the pipeline of the executing processor. Instead, read versioning allows a thread to store a version number locally without invalidating any cache lines on other processors' caches. LogTM, on the other hand, leverages the coherence protocol itself to implement a read lock using state in the cache or processor (e.g. R/W bits or signatures). Threads can acquire read locks without performing any additional memory operations and without triggering cache invalidations.

Dice and Shavit support Saha and Ennals' claim that lock-based STMs are more efficient than non-blocking STMs, but argue that late acquisition of write locks performs better under contention. Although they agree with Saha's findings that eager version management is more efficient, they argue that the improvement in concurrency from acquiring locks at commit is more compelling. Based on these findings, they propose Transactional Locking, which uses lazy version management and write locking at commit. Transactional Locking outperforms Ennal's STM on benchmarks with significant contention. Dice and Shavit, however, do not consider write set prediction or alternate conflict resolution policies, which I have shown improve the performance of LogTM. Because McRT and Ennal's STM, like LogTM, use *encounter-time* *locking* (the STM equivalent of eager conflict detection), one might expect that write set prediction and careful contention management to provide similar benefits for those systems as well.

7.3 Hardware-Software Hybrid Transactional Memory

Other researchers have proposed hybrids between software and hardware transactional memory systems. Such hybrid systems offer the possibility of a gradual migration to support for HTM by allowing the same program to take advantage of transactional memory hardware if it is present, but still run correctly otherwise. In addition, these hybrid schemes allow smaller, simpler transactions to run in hardware, while providing compatibility with software transactions, which can be used to run transactions not supported by the hardware.

7.3.1 Transactional Lock Removal

Speculative Lock Elision (SLE) [64] and Transactional Lock Removal (TLR) [65] are hardware optimizations that allow shared-memory multiprocessors to execute lock-based critical sections in a transactional manner without explicitly executing lock and release operations. TLR & SLE provide significant performance benefits to multiprocessors running conventional lock-based programs. Since using locks is the most common method of synchronization for shared-memory programs, implementing SLE & TLR will provide a speedup on a wide range of existing software. Although the applicability of TLR and SLE to existing software is certainly appealing to computer designers, their impact is ultimately limited by the fact that they do not provide the programmer a better way to write concurrent programs.

Speculative Lock Elision is based on the observation that the initial lock acquire and final lock release of a typical critical section protected by test and set locks form a temporally silent pair. That is, the release undoes the effect of the acquire. As a result, if all the actions of a critical section are observed to occur simultaneously—as they should to preserve critical-section behavior—there is no evidence that the acquire and release ever occurred. SLE uses processor speculation to elide the lock operations and perform short critical sections optimistically. If there is a conflict (e.g. the lock is contended), the speculative critical section is squashed. The processor can then either retry the critical section speculatively, or revert to conventional operation and perform the lock acquire. SLE reduces the overhead of lock operations and can eliminate unnecessary serialization of critical sections. SLE, however, does not improve performance in the presence of contention. In that case, critical sections must be serialized, and SLE can lead to increased overhead [64].

TLR extends SLE with a mechanism for serializing critical sections efficiently without reverting to the original lock actions. TLR detects lock acquire and release operations and treats code between them as a transaction. As with SLE, TLR elides the acquire and executes the transaction speculatively. TLR, however, does not blindly squash its speculative execution when a conflict is detected. Instead, a TLR-enabled processor can defer a memory request until it finishes its current transaction. Naturally, deferring memory requests can lead to conflicts between processors running transactions with overlapping data sets. Rajwar and Goodman outline a starvation-free conflict resolution algorithm based on local timestamps. In their scheme, each transaction is assigned a local timestamp. Each processor updates its timestamp at the end of each successful TLR transaction, by choosing a value later than the timestamp of any request it deferred during the last transaction. During a conflict, the transaction with the lower sequence number wins, and the other is squashed. Transactions maintain their sequence numbers so that each transaction will eventually become the oldest in the system and win all remaining conflicts [65].

Unlike LogTM and other HTMs, SLE and TLR do not change the programming model. Programmers must still create a locking mechanism that will correctly synchronize their programs and avoid deadlock. SLE and TLR, in particular, may alleviate common performance problems with locks, e.g. overly conservative synchronization. But, because the programming model is still lock-based, SLE and TLR do not address the problems associated with lock composition. Because the programs work with locks, however, SLE and TLR can improve the performance of existing non-transactional codes, which HTMs cannot do.

7.3.2 Hybrid Transactional Memory

Hybrid Transactional Memory (HyTM) aims to combine the speed of HTM with the robustness of STM [18]. In addition, HyTM promises a smoother adoption path for transactional memory support in both languages and hardware. HyTM proponents argue that programmers will adapt to transactions only if transactional memory is both supported on their current hardware (STM) and if transactional memory will provide performance benefits in the long run (HTM). Meanwhile, hardware designers will be loath to spend precious hardware resources on structures that do not speed up current applications. With HyTM, programmers can begin developing transactional memory applications immediately, which they will not have to re-write to take advantage of the HTM capabilities of future hardware.

Damron et al. propose two HyTM schemes that ensure compatibility between software and hardware transactions. The first scheme maintains a single global counter that tracks the number of currently executing software transactions. Each software transaction updates the counter when it begins and decrements the counter when it commits or aborts. Each hardware transaction reads the counter, which then becomes part of the transaction's read set. Therefore, any software transaction that begins will abort all active hardware transactions when it updates the counter. Checking the software transaction counter requires very little overhead for hardware transactions, but as soon as any transaction runs as a software transaction, all other transactions must do so as well.

HyTM has the advantage that little—in fact no—hardware support is necessary to execute transactional memory. The hardware support the authors propose is also simpler and easier to implement than LogTM. That simplicity, however, comes a price. Damron et al. report that LogTM significantly outperforms HyTM on several benchmarks [53].

7.3.3 Virtual Transactional Memory

Virtual Transactional Memory (VTM) [66] is a combined hardware/software solution to virtualize a cache-based HTM. VTM handles transaction conflict detection and version management after cache evictions, paging, and context switches. Like LogTM, VTM detects conflicts eagerly, but VTM performs version-management lazily—preventing transactions from updating shared memory locations until commit. Also like LogTM, VTM implements transactions with a combination of hardware and software. In VTM, however, some transactions are executed completely in hardware, and others completely in software, or low-level PAL or micro-code. In contrast, LogTM executes some aspects of all transactions in hardware (e.g., conflict detection) and other aspects in software (e.g., transaction rollback).

7.3.4 Bulk

Ceze et al. developed Bulk to support transactional memory with arbitrary-sized transactions and thread level speculation with large tasks [11]. Bulk encodes read and write sets in *signatures*. Signatures, like Bloom filters [7], over approximate membership in read and write sets, allowing false positives (incorrectly reporting membership), but not false negatives (incorrectly reporting non-membership).

Bulk implements HTM using lazy version management and lazy conflict detection like TCC. In Bulk, however, a processor commits a transaction by broadcasting its write signature instead of its entire write set. Similarly, when processors receive commit messages, they compare the received write signature against their own read and write signatures. As in TCC, if a processor detects a possible conflict (a non-null intersection), it aborts its transaction.

Because signatures can encode any number of addresses, transactions in Bulk can access any number of cache blocks without serializing transactions. Cache lines touched in a transaction may be evicted silently if they do not contain modified data. Any data modified in a transaction, however, must be saved to a private overflow area in memory before they are evicted from the cache. Values in the overflow area are only copied back to their main memory location on commit.

Also, unlike TCC, Bulk allows non-transactional operation. To detect conflicts between transactions and non-transactional memory accesses, processors check their read and write signatures for potential conflicts on every incoming coherence request. This practice requires that all coherence requests are compared against all active signatures, effectively requiring a broadcast-based coherence protocol.

7.3.5 Page-Granularity Transaction Virtualization

Several researchers have also proposed using the virtual memory paging mechanism to provide version management and conflict detection for transactions that overflow a cache-based HTM [14, 17]. Of these, LogTM is most similar to Chuang et al.'s Page-Based Transactional Memory (PTM). One variant of PTM, Copy-PTM, uses eager version management like LogTM. Unlike LogTM, however, the PTM mechanism is evoked on the eviction of any cache line accessed in an active transaction. Furthermore, the mechanism has a high overhead—the first cache eviction to a given page in a transaction triggers a copy of the entire page.

7.4 Speculative Multithreading

Speculative multithreading [23, 24, 76, 79, 81] or thread-level speculation, breaks a single thread of computation into multiple tasks, which, like transactions, may be executed in parallel if their data accesses do not conflict. Speculative tasks and transactions have three important differences. First, speculative tasks must appear to execute in a pre-defined order whereas transactions may execute in any sequential order. Second, because speculative tasks are part of a single thread, they share register values as well as memory. Finally, because the threading is transparent to the programmer, there is no need to support a software driven abort, or rollback in thread-level speculation.

Garzaran et al. introduce a taxonomy of speculative threading approaches, which they apply both to their own work and to other proposals [23]. Schemes are first divided into Architectural Main Memory (AMM) and Future Main Memory (FMM) categories. AMM mechanisms prevent speculative data from reaching memory. The output of a given task must be buffered until the task is ready to commit. In FMM schemes, tasks write directly to memory and maintain a backup copy of all data that are speculatively overwritten. In general, FMM schemes allow speculative tasks to be larger, but suffer longer delays from squashing tasks. AMM systems are divided into Eager AMM and Lazy AMM. Eager AMM systems merge all task updates with memory at task commit. Lazy AMM relies on normal cache write backs to merge updates with main memory, but must track which processor currently has the most recent copy of any given datum. Speculative multithreading architectures are further divided into single-task and multi-task architectures. Multi-task architectures allow a single processor to operate on more than one speculative task at a time. Both single-task and multi-task architectures can support either a single or multiple version of a given memory location. Using their taxonomy, LogTM is single-T FMM.

7.4.1 Software Logging in Thread-Level Speculation

Of the many thread-level speculation schemes, LogTM is most similar to the designs developed by the Iacoma group at the University of Illinois. Their system use Future Main Memory (FMM)—the thread level speculation equivalent of eager version management. The group presented two systems using software and hardware logging to rollback speculative state.

The first of these schemes was that presented by Zhang et al. [88, 89], which performs version management using a hardware-controlled undo log. The undo log is physically distributed throughout the various nodes in the system. Entries are added to the log by the directory controller at the home node of an array element. A processor executing a task (loop iteration) that updates an element first sends a protocol message to the home directory controller for the element. That controller then copies the value of that element (possibly generating and sending other protocol messages to other nodes) to a free entry in the local portion of the undo log.

Garzaran et al. developed the software logging scheme to reduce hardware complexity [24]. The compiler adds instructions to save the previous value of each memory location the task overwrites to the undo log. Software only controls the management of the undo log. Special hardware is still required to detect conflicts between tasks. In this case, the authors propose adding special memory instructions to allow these compiler-inserted instructions to access the task IDs. Each log record includes the ID of the task that wrote the stored version and the ID of the task that wrote the log record. Software logging performs well for applications with few squashes due to inter-task dependences—about 10% slowdown compared to a hardware-only implementation. Two drawbacks of this approach are (1) pollution in the cache from log-management and (2) cascaded rollbacks.

Chapter 8

Conclusion

Although only time will tell if transactional memory will ultimately succeed, there is already evidence that it will play some role in the future of parallel processing. Transactional memory is already gaining momentum with programmers. For example, all three of the languages developed for the High-Productivity Computer Systems challenge [62] include an atomic construct [12, 63]. Similarly, while thus far no mainstream processors have included support for HTM, a wide and growing body of research continues to demonstrate its potential.

In order for processor makers to adopt HTM, however, they must be convinced that its benefits—ease of programming or better performance—justify the expense required to build it. The key for researchers investigating transactional memory, therefore, is to find implementations of transactional memory systems that support a convenient, unrestricted transactional execution model while at the same time balancing transactional memory performance and implementation cost.

After developing and evaluating LogTM, I am ever more convinced that the only way to achieve this in a transactional memory system is through hardware/software cooperation. Supporting an unrestricted transactional memory interface, including support for very large transactions and virtualization events, is impractical, if not wasteful, in hardware. But, pure STM seems unlikely to ever match the performance of HTM or hardware-accelerated transactional memory. Leveraging cache coherence provides hardware implementations an enormous advantage over their software counterparts, especially for conflict detection. LogTM and its extensions have shown that transactional memory need not be invisible to software, but can, in fact, incorporate software in its implementation.

My experience with LogTM also leads me to believe that using eager version management and eager conflict detection is a viable approach for implementing transactional memory. That approach is only effective if abort rates are low. But, the abort rate can be controlled by the conflict detection and resolution policies. The combination of the Requester Stalls conflict resolution policy and write set prediction *makes* aborts rare. Eager version management and LogTM are particularly appealing when large transactions are considered because LogTM's version management is virtualized by default.

Much of the success of LogTM is due to the fact that it exposes transaction state—the version information in the log—to software. Similarly, Nested LogTM, LogTM-SE and LogTM-SE adapted for software contention management all expose more of the details of transactional memory to various software components, which allows those systems to be both more useful and cheaper to implement than the original LogTM.

References

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, 2006.

[2] Ardsher Ahmed, Pat Conway, Bill Hughes, and Fred Weber. AMD Opteron Shared Memory MP Systems. In *Proceedings of the 14th HotChips Symposium*, August 2002.

[3] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint Processing and Recovery: An Efficient, Scalable Alternative to Reorder Buffers. *IEEE Micro*, 23(6), Nov/Dec 2003.

[4] Alaa R. Alameldeen and David A. Wood. Addressing Workload Variability in Architectural Simulations. *IEEE Micro*, 23(6):94–98, Nov/Dec 2003.

[5] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, February 2005.

[6] Ernest Artiaga, Nacho Navarro, Xavier Martorell, and Yolanda Becerra. Implementing PARMACS Macros for Shared Memory Multiprocessor Environments. Technical report, Polytechnic University of Catalunya, Department of Computer Architecture Technical Report UPC-DAC-1997-07, January 1997.

[7] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[8] Colin Blundell, E Christopher Lewis, and Milo M.K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[9] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[10] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[11] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, June 2006.

[12] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 2007.

[13] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[14] Weihaw Chuang, Satish Narayanasmy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Osvaldo Colavin, and Brad Calder. Unbounded Page-Based Transactional Memory. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[15] Duk Chun and Shabbir Latif. MIPS R4000 Synchronization Primitives. Technical Report AP004, MIPS Technologies, Inc., April 1993.

[16] JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, February 2006.

[17] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Hassan Chafi, Brian D. Carlstrom, Travis Skare, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in Transactional Memory Virtualization. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[18] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchango, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[19] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[20] Robert Ennals. Efficient Software Transactional Memory. Technical Report IRC-TR-05-051, Intel Research Cambridge, January 2005.

[21] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[22] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling Long-Running Activities as Nested Sagas. *IEEE Bulletin of the Technical Committee on Data Engineering*, 14(1):14–18, 1991.

[23] María Jesús Garzarán, Milos Prvulovic, Victor Viñals, José María Llabería, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, February 2003.

[24] María Jesús Garzarán, Milos Prvulovic, Victor Viñals, José María Llabería, Lawrence Rauchwerger, and Josep Torrellas. Using Software Logging to Support Multi-Version Buffering in Thread-Level Speculation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[25] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[26] J. Gray, R. Lorie, F. Putzolu, and I. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. In *Modeling in Data Base Management Systems, Elsevier North Holland, New York*, 1975.

[27] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, September 1981.

[28] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *International Conference on Parallel Processing (ICPP)*, volume I, pages 312–321, 1990.

[29] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[30] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, October 2003.

[31] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, June 1991.

[32] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Twenty-Second ACM Symposium on Principles of Distributed Computing, Boston, Massachusetts*, July 2003.

[33] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. Technical Report Technical Report 92/07, Digital Cambridge Research Lab, 1992.

[34] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[35] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, February 2001.

[36] Tim Horel and Gary Lauterbach. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3):73–85, May/June 1999.

[37] IBM Corporation. *Book E: Enhanced PowerPC Architecture, version 0.91*, July 21, 2001.

[38] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[39] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, March-April 2003.

[40] Tom Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 105–112, 1986.

[41] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):29–25, Mar/Apr 2005.

[42] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, pages 213–226, June 1981.

[43] Leslie Lamport. The Mutuall Exclusion Problem: Part I, A Theory of Interprocess Communication. *J. ACM*, 33(2):313–326, 1986.

[44] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[45] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[46] Scott Lie. Hardware Support for Unbounded Transactional Memory. Master's thesis, MIT, May 2004.

[47] Tim Lindholm and Frank Yellin. *The JavaTM Virtual Machine Specification*. Sun Microsystems, 1999.

[48] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[49] V.J. Marathe, W.N. Scherer III, and M.L. Scott. Adaptive Software Transactional Memory. In Pierre Fraigniaud, editor, *Distributed algorithms*, volume 3724 of *Lecture Notes In Computer Science*, pages 354–368, September 2005.

[50] Milo M. K. Martin et al. Protocol Specifications and Tables for Four Comparable MOESI Coherence Protocols: Token Coherence, Snooping, Directory, and Hammer. http://www.cs.wisc.edu/multifacet/theses/milo_martin_phd/, 2003.

[51] Austen McDonald, JaeWoong Chung, Brian Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, June 2006.

[52] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *Readings in Database Systems*, pages 251–285. Morgan Kaufmann Publishers, 1998.

[53] Mark Moir. Hybrid Transactional Memory. Presented at the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2006.

[54] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-Based Transactional Memory. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, pages 258–269, February 2006.

[55] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting Nested Transactional Memory in LogTM. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, October 2006.

[56] Andreas Moshovos, Gokhan Memik, Babak Falsafi, and Alok Choudhary. JETTY: Filtering Snoops for Reduced Power Consumption in SMP Servers. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, January 2001.

[57] J. Eliot B. Moss. *Nested transactions: an approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[58] J. Eliot B. Moss. Nesting Transactions: Why and What Do We Need? TRANSACT Keynote Address, June 2006.

[59] J. Eliot B. Moss. Open Nested Transactions: Semantics and Support. In *Workshop on Memory Performance Issues*, February 2006.

[60] J. Eliot B. Moss and Antony L. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *SCOOL Workshop*, October 2005.

[61] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. MIT Press, 1972.

[62] DARPA Information Processing and Technology Office. High Productivity Computer Systems. http://www.highproductivity.org/.

[63] Fortress Project. Fortress Project Home. http://fortress.sunsource.net/.

[64] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2001.

[65] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[66] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[67] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmet Witchel. MetaTM/TxLinux: Transactional Memory for an Operating System. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[68] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw Hill, 2000.

[69] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 2006.

[70] M. Satyanarayanan and Dileep Bhandarkar. Design Trade-Offs in VAX-11 Translation Buffer Organization. *Computer*, 14(12):103– 111, December 1981.

[71] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, July 2005.

[72] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Fourteenth ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada*, pages 204–213, August 1995.

[73] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[74] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[75] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[76] G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[77] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology, Systems, & Applications*, 1(4):58–71, November 1993.

[78] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[79] Jean-Yuan Tsai and Pen-Chung Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, October 1996.

[80] Andrew Tucker, Bart Smaalders, Dave Singleton, and Nicolai Kosche. Method and apparatus for execution and preemption control of computer process entities, August 1999. U.S. Patent 5,937,187.

[81] T.N. Vijaykumar, Sridar Gopal, James E. Smith, and Gurindar Sohi. Speculative Versioning Cache. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1305–1317, December 2001.

[82] David L. Weaver and Tom Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[83] Gerhard Weikum and Hans-Jorg Schek. *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*. Morgan Kaufmann, 1992.

[84] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[85] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[86] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[87] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory

from Caches. In *Proceedings of the Thirteenth IEEE Symposium on High-Performance Computer Architecture*, pages 261–272, February 2007.

[88] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, February 1998.

[89] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors,. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, January 1999.

[90] Craig Zilles and David H. Flint. Challenges to Providing Performance Isolation in Transactional Memories. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.