IMPROVING THE PRACTICALITY OF TRANSACTIONAL MEMORY

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

Woongki Baek
December 2010

This dissertation is online at: http://purl.stanford.edu/mr457jn9471

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

Christoforos Kozyrakis, Primary Adviser

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

Oyekunle Olukotun, Co-Adviser

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

Subhasish Mitra

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Transactional Memory (TM) simplifies parallel programming by transferring responsibility for concurrency management from the programmer to the system. Nevertheless, a number of challenges must be addressed for TM to be widely adopted. This dissertation presents work towards improving the practicality of TM across three dimensions.

The first challenge addressed is that of integrating TM with popular models for high-level parallel programming. The need for high-level programming environments for TM is ever increasing to improve the programmability of TM. To address this challenge, we propose OpenTM, a programming model that integrates TM into OpenMP, a widely-used parallel programming environment. The high-level language constructs of OpenTM simplify TM programming by abstracting low-level programming issues such as manual instrumentation of TM barriers and by allowing high-level management of TM programming options. OpenTM aims to deliver good performance with simple and portable high-level code.

The second challenge addressed is that of supporting nested parallelism within transactions. To achieve the best possible performance on large-scale parallel systems, it is important to fully exploit the parallelism available at all levels. To address this challenge, we propose two TM systems with support for concurrent nesting: (1) NesTM, a software TM (STM) system that supports nested parallel transactions solely in software and (2) Filter-accelerated Nested TM (FaNTM), a hybrid TM system with lightweight hardware support for nested parallel transactions. NesTM presents a good model for concurrent nesting, and FaNTM provides a fast and practical implementation. In particular, the hardware filters of FaNTM provide continuous

and nesting-aware conflict detection, which effectively eliminates the excessive runtime overheads of software nested transactions. In contrast to a full hardware TM approach, FaNTM simplifies hardware by decoupling nested transactions from the hardware caches.

The third challenge addressed is that of verifying the correct operation of TM systems. Since TM systems take responsibility for managing shared state on behalf of the applications, it is important to rigorously check their correct operation. To address this challenge, we propose ChkTM, a flexible model checking environment to check the correctness of various TM systems. ChkTM models STMs close to the implementation level to reveal as many bugs as possible. Using ChkTM, a subtle correctness bug in a widely-used STM implementation has been found. In addition, the serializability of several TM systems has been checked.

Acknowledgements

First of all, I would like to thank my adviser, Christos Kozyrakis, for his sincere help and support during my graduate studies at Stanford. He is not only a technically solid researcher but also a kind and patient adviser. Definitely, it would have not been possible for me to complete this dissertation without his sincere guidance. I truly hope that we could have more opportunities for collaborating on exciting research projects even after my graduation.

I would also like to thank other professors who have been greatly influential to me during my graduate studies at Stanford. First, I am truly grateful to Kunle Olukotun, the co-principal investigator of the Transactional Coherence and Consistency (TCC) research group at Stanford. His insightful guidance has helped me address technical challenges. I am also sincerely thankful to Subhasish Mitra for his valuable feedback that has helped me improve this dissertation.

I would also like to thank the TCC group members. Without their valuable support, it would have been very difficult for me to complete this work. In particular, my work has significantly relied on the architectural simulator and applications developed by Austen McDonald, Chi Cao Minh, JaeWoong Chung, and Jared Casper. I am also truly grateful to Nathan Bronson for sharing his brilliant insights that have greatly helped me address technical challenges. In addition, I would like to thank my colleagues in the research group of Prof. Kozyrakis.

I would like to express my sincere gratitude and love to my family. My parents have guided me with their priceless wisdom and have been always praying for me with their sincere love. My brother and sister-in-law have given me precious advices

and strength. I miss my dear nephew and niece so much. I am truly blessed to have all of them as my family.

Finally, I would like to give my sincere thanks to God with all my heart.

My graduate studies have been funded by a Samsung Scholarship and an STMicroelectronics Stanford Graduate Fellowship.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The performance of microprocessors has been steadily improving over the years. This is mainly due to the advance in device technologies that enable exponentially increasing transistor counts with higher clock frequencies and the use of techniques (e.g., wide issue, out-of-order execution) that exploit the instruction-level parallelism (ILP) available in instruction streams. Unfortunately, however, improving single-threaded performance based on these conventional techniques has recently faced fundamental challenges such as the limited amount of ILP in most applications [98], unacceptable power consumption [7], and drastically increased design complexity [101].

As a result, chip multiprocessors (CMPs) are becoming the norm for a wide range of application domains including datacenter and embedded computing [1,23,42,55,57, 58,73,102]. Multiple simpler processor cores in CMP systems reveal new opportunities for improving performance in a power-efficient manner by utilizing the thread-level parallelism (TLP) available in parallel applications [76, 77].

1.1 The Current Practice of Parallel Programming

To achieve scalable performance on CMPs, however, programmers must deal with the complexity of parallel programming. To implement parallel applications, programmers should create and synchronize parallel tasks. To date, locks have been widely

used as a low-level synchronization primitive. Locks synchronize shared-memory accesses performed by concurrent threads via mutual exclusion. If a thread acquires a lock for certain data, it obtains exclusive ownership for accessing the data. Other threads that attempt to acquire the same lock (and access the same data) are blocked until the thread releases the lock.

The difficulty of lock-based programming mainly stems from the undesirable trade-off between programming simplicity and scalable performance. While coarse-grain locking is relatively easy to use, it may drastically degrade performance by introducing unnecessary serialization among concurrent threads. Fine-grain locking often leads to higher performance by allowing greater concurrency. With a greater number of (fine-grain) locks, however, programming complexity is significantly increased, causing various problems such as deadlock, priority inversion, and convoying [63]. Fine-grain locking may also degrade performance in the absence of contention.

Furthermore, lock-based parallel programming makes it difficult to compose multiple program modules. Even with individual code blocks correctly synchronized with locks, they can cause various correctness issues when they are composed as the inconsistent intermediate state can be unsafely exposed [49]. Note that software composability issues exist regardless of the lock-granularity (e.g., coarse- or fine-grain locking) used for synchronization.

1.2 Transactional Memory and Challenges

Transactional Memory (TM) [53] has been proposed to simplify parallel programming. TM addresses the difficulty of lock-based parallel programming by allowing programmers to simply declare certain code blocks as *transactions*. TM guarantees that user-defined transactions execute in an *atomic* and *isolated* manner with respect to the other code blocks. A large number of TM systems have been proposed, based on hardware [10, 19, 21, 46, 69, 79], software [37, 50, 52, 65, 81, 83], or hybrid techniques [26, 59, 82].

While TM is a promising technology to simplify parallel programming, there still remain important challenges that must be addressed to make TM more practical

and widely adopted in mainstream parallel programming. First, TM should be integrated into high-level programming environments to improve its programmability by abstracting low-level details (e.g., managing concurrent threads) of TM programming. The current practice of TM programming is primarily based on libraries. They typically include low-level TM library functions to define transaction boundaries, instrument shared memory accesses, and manage the runtime system. This library-based approach might be sufficient for small-scale software projects. It is, however, inadequate for large-scale software projects by non-expert developers as library-based code is difficult to comprehend, maintain, and scale. Furthermore, the library-based approach reduces the opportunities for compiler optimizations [5]. Thus, it is important to integrate TM into popular, high-level programming environments.

Second, TM should be extended to efficiently support nested parallelism. Most TM systems, so far, have assumed that the code within transactions is sequentially executed. Real-world applications, however, often include *nested parallelism* in various forms such as calls to parallel libraries, nested parallel loops, and recursive function calls [91]. To achieve the best possible performance on systems with the increasing number of cores, it is important to fully exploit the parallelism available at all levels. In this spirit, several popular programming environments that do not use transactions have already incorporated nested parallelism [2, 90]. Therefore, it is important for TM to efficiently support the case of nested parallelism.

Third, a flexible model checking environment is required to check the correctness of a variety of TM systems. On behalf of transactional applications running on it, TM takes responsibility for managing all shared-memory accesses. Therefore, it is critical to ensure that the underlying TM system is correct. At the same time, extreme attention should be paid to the performance of the system. Consequently, subtle but fast TM implementations are favored over simpler ones, even though it makes the resulting implementations difficult to prove correct. Because of the central position of TM, the severity of any bug is drastically magnified. With the difficulty of formally proving the correctness of TM, the importance of a model checking environment for TM is ever increasing.

1.3 Contributions

In this dissertation, we present work towards improving the practicality of TM across three dimensions. Specifically this dissertation makes the following contributions:

- **Improving the programmability of transactional memory**

  We propose *OpenTM*, an application programming interface (API) for parallel programming with atomic transactions [14]. We define the set of extensions to OpenMP, a popular programming environment for shared-memory systems, to support both non-blocking synchronization and speculative parallelization based on TM. OpenTM provides language constructs to integrate memory transactions with OpenMP constructs such as parallel loops and sections. In addition, the advanced language constructs of OpenTM address issues such as nested transactions, conditional synchronization, scheduling, and contention management. We describe an OpenTM implementation for the C programming language. Using a variety of transactional applications, we demonstrate that the OpenTM code is simple, compact, and scales well.

- **Supporting nested parallelism in transactional memory**

  We propose two TM systems with support for nested parallelism within transactions. First, we present *NesTM*, a software TM system that supports nested parallel transactions solely in software [12]. We describe the design and implementation of NesTM and quantify the performance of nested parallel transactions on NesTM. Second, we present *filter-accelerated, nested transactional memory* (*FaNTM*) that provides practical support for nested parallel transactions using hardware filters [13]. NesTM presents a good model for concurrent nesting, and FaNTM provides a fast and practical implementation. Specifically, FaNTM extends its baseline hybrid TM system to implement nesting-aware conflict detection and data versioning in a performance- and cost-effective manner. We also describe subtle correctness and liveness issues that do not exist in the non-nested baseline TM and propose solutions to address the problems. To

showcase the effectiveness of FaNTM, we quantify the performance of FaNTM across multiple use scenarios using a variety of transactional applications.

- **Implementing a model checker for transactional memory**

  We propose *ChkTM*, a flexible model checking environment for checking the correctness of TM systems [11]. ChkTM aims to model TM systems close to the implementation level to reveal as many potential bugs as possible. In ChkTM, we model several TM systems including a widely-used STM. We also describe a case study in which we actually found a subtle correctness bug in the current implementation of the STM. We also perform an in-depth, quantitative analysis on ChkTM to understand the practical issues in checking the correctness of TM systems.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 reviews transactional memory. Chapter 3 presents OpenTM, an application programming interface for parallel programming with transactions. Chapter 4 describes NesTM, a software TM system that supports nested parallel transactions solely in software. Chapter 5 presents filter-accelerated nested transactional memory (FaNTM) that provides practical support for nested parallel transactions using lightweight hardware. Chapter 6 describes ChkTM, a flexible model checking environment for checking the correctness of TM systems. Finally, Chapter 7 concludes this dissertation.

Chapter 2

Transactional Memory

This chapter introduces Transactional Memory (TM) and reviews TM design space, programming models, and implementation options. We refer readers to Section 1.2 for the motivations for the three TM challenges (i.e., improving programmability, supporting nested parallel transactions, and checking the correctness of TM systems) that will be discussed in the subsequent chapters. We also refer readers to previous proposals for the other TM challenges such as virtualization [31, 32, 78] that are important but out of the scope of this dissertation.

2.1 Overview of Transactional Memory

With lock-based parallel programming, programmers must consider not only where synchronization is required but also how it is implemented. Programmers should associate shared data with locks and carefully follow the established association to ensure correctness. Programmers also need to strictly follow a consistent order in which locks are acquired and released to avoid possible deadlock scenarios. Furthermore, programmers should determine the proper lock granularity to achieve scalable performance without sacrificing programmability.

In contrast, TM simplifies parallel programming by transferring concurrency control from the user to the system. With TM, programmers simply declare code blocks

```
atomic {
  if (tree_1.contains(x)) {
    tree_1.remove(x);
    tree_2.insert(x);
  }
}
```

Figure 2.1: A code example of using a transaction.

as *transactions* if synchronization is required. It is the responsibility of the underlying TM system to guarantee correctness and liveness when executing the application code. Figure 2.1 shows an example of how a transaction can be used to remove an item from a tree and insert it into another tree in an atomic manner.

The common-case execution of a transaction is as follows. After taking a checkpoint of the execution context, the transaction speculatively executes. During the execution, the memory addresses that are read or written by the transaction are summarized in the *read* or *write set* of the transaction. To provide *data versioning*, the write set also holds old or new values for the memory objects written by the transaction. If the transaction successfully reaches to the end of the atomic block, it *commits* by making all its updates permanently visible outside the transaction. Otherwise, the transaction *aborts* by rolling back all the side effects and restarts its execution.

TM should satisfy the following properties: *atomicity*, *isolation*, and *serializability*. Atomicity requires that all or none of the instructions in a transaction appear to occur outside the transaction. For example, a transaction executing the code block in Figure 2.1 appears to have performed all or none of the tree operations with respect to other transactions. With the isolation property, the intermediate state of a transaction is externally invisible until it commits. For instance, no other transactions are allowed to observe the intermediate state of the trees while the trees are being updated by a transaction executing the code block in Figure 2.1. With the serializability property, the outcome of transactions concurrently executed is equivalent to that of the same transactions executed serially. For example, if the code block in Figure 2.1

is concurrently executed by multiple transactions, the final state of the trees should be equivalent to that of the trees updated by the same transactions that are serially executed.

A correctness criterion commonly used for TM systems is *conflict serializability* [41, 88]. Two transactions *conflict* if they access the same memory location *and* at least one of them performs a write operation to the memory location. To provide fine-grain conflict detection, TM compares the read and write sets of a transaction with those of other concurrent transactions. If no conflicts have been detected, the transaction can successfully commit. Otherwise, the transaction aborts and retries.

Beside the programming simplicity with large atomic blocks, TM provides additional advantages. First, TM achieves good performance through optimistic and fine-grain concurrency control. With coarse-grain locks, code is executed in a pessimistic manner in which threads are blocked even when there is no true data dependency. In contrast, with TM, transactions are executed in a speculative (optimistic) manner as if they do not conflict with other transactions. Conflicts occur only when there is a true data dependency with other transactions.

In addition, TM provides *safe composability* of code blocks. As shown in Figure 2.1, TM allows for multiple tree operations to be performed atomically without exposing any inconsistent intermediate state by enclosing all the operations with a transaction. In contrast, if two operations that are correctly synchronized with locks are composed, they can cause various correctness issues as the inconsistent intermediate state can still be exposed [49].

2.2 TM Design Space

This section reviews major dimensions in TM design space such as data versioning and conflict detection.

2.2.1 Data Versioning

A memory location written by a transaction is associated with two values: *speculative* and *non-speculative*. If a transaction successfully commits, all the data speculatively written by the transaction become non-speculative data that are externally visible. If a transaction aborts, all the speculative data of the transaction must be safely discarded to guarantee the atomicity and isolation properties. Data versioning can be performed either eagerly or lazily.

With *eager data versioning*, each transaction directly updates a memory location when it speculatively performs a write operation. The transaction records the old value of the memory location in its undo log. Eager data versioning allows for fast transactional commits because the speculative data have been already updated in place, which eliminates the need for traversing the write set to update the memory locations at the commit time. In contrast, transactional aborts are slow because transactions need to traverse their write sets to rollback the speculatively written values.

With *lazy data versioning*, updates to memory locations are deferred until each transaction reaches to its commit point. The values speculatively written by the transaction are buffered in its write buffer. Lazy data versioning allows for fast transactional aborts because each transaction can simply reset its write buffer without traversing its write set to rollback the speculatively written data. However, lazy data versioning has the following performance disadvantages. First, it leads to slower transactional commits because each transaction should traverse its write set to apply the speculatively written data to main memory. Second, it may lead to slower transactional reads because each transaction should first search its write buffer to check the speculatively written values.

2.2.2 Conflict Detection

TM systems can perform conflict detection in a pessimistic or optimistic manner. With *pessimistic conflict detection*, conflicts are checked on every transactional memory access. Each memory access is allowed to complete only if it does not conflict

with any other concurrent transactions. By detecting conflicts early, the pessimistic scheme may reduce the amount of work performed by the transactions that eventually abort, thus potentially improving overall performance.

In contrast, *optimistic conflict detection* checks conflicts when a transaction attempts to commit. It avoids the overhead of checking conflicts on every transactional access. In addition, optimistic conflict detection allows for more serializable schedules to successfully commit compared to the pessimistic scheme. Due to the deferred (i.e., optimistic) conflict detection, however, a large amount of work can be wasted, which may degrade overall performance.

2.3 TM Programming Models

There are a number of TM programming models to consider when implementing TM systems and applications. This section reviews a few key concepts such as how transactions are isolated, how transactional barriers are used, and how transactional nesting is handled.

2.3.1 Weak and Strong Isolation

Unlike the database systems [41, 88], TM should handle accesses to shared data performed by non-transactional code. Therefore, it is important for programmers to understand how transactions interact with non-transactional code in the underlying TM system. With *strong isolation*, transactions are isolated from both transactional and non-transactional memory accesses [20, 61]. Conceptually, with strong isolation, every non-transactional memory access appears to execute as a single transaction. In contrast, with *weak isolation*, transactional semantics are only guaranteed among transactional code.

Weak isolation makes it more difficult to write correct transactional applications. Programmers must ensure that non-transactional code does not access the same data with transactional code in parallel regions. Otherwise, subtle correctness issues may occur [38, 61, 86]. One example is the *non-repeatable read* anomaly where multiple

```
atomic {
 if (t->root==NULL) {
  t->root = node;
 }
}
```

(a) Implicit

```
TxStart();
if (TxLoad(&t->root)==NULL) {
 TxWrite(&t->root, node);
}
TxCommit();
```

(b) Explicit

Figure 2.2: Comparison between implicit and explicit transactions.

reads from the same memory address within a transaction observe different versions of the data, which violates the isolation property [86]. Section 6.2 discusses the issues with weak isolation in more detail. In contrast, strong isolation eliminates such anomalies because transactions are isolated from both other transactions and non-transactional memory accesses. Implementing strong isolation, however, may have an unattractive performance impact on non-transactional code. Hence, some TM systems implement weak isolation to achieve higher performance at the expense of programming simplicity.

2.3.2 Implicit and Explicit Transactions

The read and write sets of transactions can be constructed in an *implicit* or *explicit* manner. Figure 2.2 illustrates how the code blocks that atomically insert an element into an empty tree can be implemented using implicit and explicit transactions. Implicit transactions require no manual instrumentation of TM barriers as the memory accesses within transactions are tracked in an implicit manner with hardware or software (e.g., compiler or dynamic binary translator) support. As shown in Figure 2.2, however, with explicit transactions, programmers should explicitly annotate the shared-memory accesses within transactions with TM barriers.

Explicit transactions have an advantage in the sense that programmers can optimize the application code based on application-level knowledge. For example, programmers may skip TM barriers when accessing thread-local or immutable variables.

However, explicit transactions dreadfully increase the burden on programmers because programmers must ensure that all the shared-memory accesses within transactions are correctly instrumented with TM barriers. If there is any missing barrier, the program behavior can be incorrect or unpredictable, exposing subtle bugs that are often hard to find and fix. In contrast, implicit transactions significantly reduce the burden on programmers because no manual instrumentation of TM barriers is required.

2.3.3 Transactional Nesting

Transactions are *nested* when an outer transaction fully encloses an inner transaction [72]. There are primarily two types of nesting: *closed* and *open*. With closed nesting, the updates made by a nested transaction become externally visible only after the outer transaction commits [72]. When a nested transaction aborts, the code in the nested transaction is re-executed without aborting the outer transaction, which often leads to improved performance. With open nesting, the updates of a nested transaction become externally visible as soon as it commits (even if the outer transaction has not committed yet) [9, 71]. When an outer transaction aborts, compensation actions should be executed to nullify the side effects made by the nested transaction. Open nesting is commonly used to allow for the execution of the system code within a user-level transaction with independent atomicity and isolation [28, 66].

Apart from the semantics of nesting, there are two design choices in how nested transactions execute [72]. A TM system that implements *linear nesting* serially executes nested transactions. In contrast, a TM system with *concurrent nesting* allows for the concurrent execution of nested transactions to fully exploit the nested parallelism available in outer transactions. Support for nested parallelism in TM is increasingly important to achieve the best possible performance on large-scale parallel systems. Chapters 4 and 5 provide detailed discussions of concurrent nesting.

2.4 TM Implementation Options

A large number of TM implementations have been proposed based on hardware, software, and hybrid techniques. This section reviews these three implementation options.

2.4.1 Hardware Transactional Memory Systems

Hardware TM (HTM) systems implement data versioning and conflict detection by modifying the hardware caches and the coherence protocol [10, 19, 21, 46, 67, 69, 79]. Data versioning is provided by either logging old values (eager) or buffering speculative values (lazy) in the cache. To track the read and write sets of a transaction, each cache line is extended with the *speculatively read* (SR) and *speculatively written* (SW) bits. If a transaction reads (or writes) a memory object, the SR (or SW) bit of the cache line holding the memory object is set. A transaction detects a conflict when it receives a conflicting memory request from other processor for a cache line in its read or write set (i.e., the SR or SW bit of the cache line is set).

The key advantages of the HTM approach are high performance and inherent strong isolation guarantees. Since hardware continuously tracks the read and write sets of transactions in a transparent manner (i.e., no software annotations), HTM systems achieve high performance. In addition, since hardware monitors every coherence message including non-transactional memory accesses, strong isolation is guaranteed without any extra performance overhead.

Despite the advantages of the HTM approach, implementing all the transactional functionality solely in hardware significantly increases hardware design and verification costs as it requires intrusive modifications of the hardware caches and the coherence protocol. Furthermore, since the hardware resources are limited, it is challenging to virtualize the key aspects (e.g., time, size) of transactional execution on HTM systems [31, 32, 78]. Due to the hardware complexity and inflexibility, major hardware vendors have been reluctant to commercialize full HTM systems.

2.4.2 Software Transactional Memory Systems

Software TM (STM) systems implement data versioning and conflict detection solely in software [37, 52, 65, 81, 83]. While a large number of STM designs have been proposed, we focus on timestamp-based, blocking STM systems because they have performance advantages and are widely used in industry-strength STM systems [37, 50, 83]. The proposed STMs can be categorized into *lazy* and *eager* STMs.

A lazy STM maintains software write buffers to implement lazy data versioning [37]. It uses a global version clock to establish serializability. Using a hashing function, each memory object is associated with a version-owner lock (voLock) that either acts as a lock or stores a version number. The version number indicates the global version clock value at the time when the memory object was updated by a committing transaction. When a transaction reads from a memory object, it first checks its write buffer. If not found, it checks the associated voLock to detect any conflict. If no conflict is detected, the transaction inserts the address of the memory object into its read set and reads the memory value. When a transaction writes to a memory object, it buffers the update into its write buffer. On commit, a transaction performs the following steps: (1) acquiring the voLocks for all the memory objects in its write set and (2) validating all the memory objects in its read set. If any of the two steps fails, the transaction aborts. Otherwise, the transaction writes back the buffered updates, increments the version number in the acquired voLocks, and releases the voLocks.

In comparison, an eager STM [25, 38] maintains software undo logs to implement eager (in place) data versioning. While the read barrier is similar to the one in the lazy STM, it does not need to check the write buffer as the memory holds the speculative data. When a transaction writes to a memory object, it first attempts to acquire the ownership for the memory object. If this acquisition fails, the transaction aborts. Otherwise, the transaction updates the memory object in place and inserts the old value into its undo log. On commit, a transaction validates all the memory objects in its read set. If this validation fails, the transaction aborts by restoring the speculatively written memory objects and releasing the acquired voLocks. Otherwise,

the transaction increments the version number in the acquired voLocks and releases them.

STMs have an advantage in the sense that they can run on existing commodity hardware systems. This is because STMs implement all the transactional functionality solely in software, requiring no specialized hardware support. It is also straightforward to virtualize transactions on STMs because they are not limited by any specific hardware resource. However, STMs incur significant runtime overheads due to the frequent use of software transactional barriers. Depending on the application, single-thread STM code runs up to 7× slower than sequential code [5, 26]. It is also challenging to implement strong isolation in STMs. To guarantee strong isolation, non-transactional memory accesses should be also instrumented with transactional barriers, which incurs additional runtime overheads. To avoid further performance degradation, many high-performance STM implementations provide weak isolation [37, 50, 83].

2.4.3 Hybrid Transactional Memory Systems

Recently, a number of TM systems have been proposed based on hybrid techniques. The proposed hybrid TM systems use both hardware and software components to implement transactional functionality [16,26,36,59,87,94]. The key design philosophy of this hybrid approach is to implement a high-performance TM system with low hardware costs. Among the proposed hybrid TM systems, we focus on SigTM [26] because it satisfies various desirable properties such as high performance, flexibility, low hardware complexity, and strong isolation. Similar to STMs, SigTM can be implemented in two different ways: *lazy* and *eager*.

Lazy SigTM [26] accelerates a lazy-versioning STM using *hardware signatures* [18, 29,92]. Hardware signatures conservatively represent the read and write sets of transactions. Hardware signatures continuously snoop coherence messages to provide conflict detection. Lazy SigTM uses a software write buffer to perform lazy versioning. The read and write barriers of lazy SigTM operate similarly to those of the lazy STM, except that software read and write sets are replaced with hardware signatures. On

commit, a transaction performs write-set validation by acquiring the exclusive ownership for all the cache lines holding the memory objects in its write buffer. If this write-set validation fails, the transaction aborts. Otherwise, the transaction successfully commits by writing back all the buffered updates.

In contrast to lazy SigTM, eager SigTM [25] uses software undo logs to perform eager data versioning. The read barrier simply inserts the address into the read signature and reads the memory value. Before updating a memory object, the write barrier attempts to acquire the exclusive ownership for the cache line holding the memory object. If this acquisition fails, the transaction aborts. Otherwise, it updates the memory object in place and inserts the old value into its undo log.

SigTM achieves significantly higher performance (30% to 280%) than STMs [26] by eliminating most of the runtime overheads of software barriers with hardware signatures. For example, the read barrier of SigTM does not need to access the complicated software data structures that represent the read sets of transactions as they are replaced by hardware signatures. In addition, with hardware signatures, some of the iteration passes over the read and write sets at the commit time can be skipped.

The hardware complexity of SigTM is significantly lower than HTMs because it does not require intrusive modifications to the hardware caches. Compared to full HTMs, SigTM implements unbounded transactions without requiring complicated hardware support. This is because SigTM transactions are not bounded by the hardware cache size as hardware signatures summarize all the TM metadata. In addition, the write-set size of a transaction is not limited by hardware resources because SigTM provides data versioning in software. Finally, SigTM inherently implements strong isolation because hardware signatures detect conflicting memory requests by monitoring all the coherence messages, even non-transactional ones. The advantages of the SigTM design motivated us to use SigTM as our baseline hybrid TM system to design a TM system with support for nested parallelism (Chapter 5).

Chapter 3

The OpenTM Transactional API

3.1 Introduction

For widespread use, it is important to integrate TM into practical and familiar programming environments. TM programming, however, has been primarily based on libraries that include special functions to define transaction boundaries, manipulate shared data, and manage the runtime system. While the library-based approach is sufficient for initial experimentation with small programs, it is inadequate for large-scale software projects by non-expert developers. Furthermore, library-based code is difficult to comprehend, maintain, port, and scale, because the programmer must manually annotate accesses to shared data. Library annotations also complicate static analysis and reduce the effectiveness of compiler optimizations [5].

To address this problem, this chapter presents *OpenTM*, an application programming interface (API) for transactional programming. OpenTM extends the popular OpenMP API for shared-memory systems [2] with the compiler directives necessary for expressing both *non-blocking synchronization* and *speculative parallelization* using memory transactions. OpenTM provides a simple and high-level interface to express transactional parallelism, identify the role of key variables, and provide scheduling and contention management hints. OpenTM code can be efficiently mapped to multiple TM systems such as hardware and software TM implementations. It can also be tuned by an optimizing compiler and dynamically scheduled by the runtime system.

The specific contributions of this work are:

- We define the set of extensions to OpenMP that support non-blocking synchronization and speculative parallelization based on transactional memory techniques. In addition to integrating transactions with the OpenMP constructs such as parallel loops and sections, the extensions address issues such as nesting, conditional synchronization, scheduling, and contention management.

- We describe an OpenTM implementation for the C programming language. The environment is based on a source-to-source OpenMP environment [95] and the transactional memory support in the GCC compiler [4]. The current OpenTM implementation produces executable code for multiple TM platforms such as hardware and software TM systems. The implementation is easy to port to any TM system that provides a basic low-level interface for transactional functionality.

- We evaluate the performance and programmability features of parallel programming with OpenTM. Using a variety of programs, we demonstrate that OpenTM code is simple, compact, and scales well.

The rest of this chapter[1] is organized as follows. Section 3.2 reviews the OpenMP API. Section 3.3 introduces the OpenTM extensions. Section 3.4 describes a prototype OpenTM implementation. Sections 3.5 and 3.6 present the quantitative evaluation. Section 3.7 reviews related work. Finally, Section 3.8 concludes the chapter.

3.2 OpenMP Overview

OpenMP is a widely-used API for shared-memory parallel programming [2]. The OpenMP specification includes a set of compiler directives, runtime library routines, and environment variables. OpenMP follows the *fork-join* parallel execution model. The *master thread* starts a program execution sequentially. When the master thread

[1]The work presented in this chapter was also published in [14].

encounters a `parallel` construct, it creates a set of *worker threads*. All workers execute the same code inside the `parallel` construct. When a work-sharing construct is encountered, the work is divided among the concurrent workers and cooperatively executed. After the end of a work-sharing construct, every thread in the team resumes execution of the code in the `parallel` construct until the next work-sharing opportunity. There is an implicit barrier at the end of the `parallel` construct, and only the master thread executes user code beyond that point.

The OpenMP memory model is shared memory with relaxed consistency [6]. All threads can perform load and store accesses to variables in shared memory. The `parallel` directive supports two types of variables within the corresponding code block: *shared* and *private*. Each variable referenced within the block has an original variable with the same name that exists outside the `parallel` construct. Memory operations to a shared variable by each thread will access the original variable. For private variables, each thread has its own private copy of the original variable.

OpenMP provides programmers with five classes of directives and routines: parallel, work-sharing, synchronization, data environment, and runtime library routines. The `parallel` directive starts a parallel region. The work-sharing directives can describe parallel work for iterative (i.e., loops) and non-iterative (i.e., parallel sections) code patterns. The following is a simple code example of a parallel loop in OpenMP:

```
#pragma omp parallel for
  for(i=0;i<n;i++)
    b[i]=(a[i]*a[i])/2.0;
```

Synchronization constructs such as `critical` allow threads to coordinate on shared-memory accesses. Data environment primitives describe the sharing attributes of variables accessed in parallel regions. Finally, runtime routines specify a set of functions and variables used for runtime optimizations, synchronization, and scheduling. More detailed specification and tutorials for OpenMP are available in [2].

3.3 The OpenTM API

OpenTM is designed as an extension of OpenMP to support non-blocking synchronization and speculative parallelization based on transactional memory techniques. Therefore, OpenTM inherits the key features of OpenMP such as the execution model, memory semantics, language syntax, and runtime constructs. Every OpenMP program is a legitimate OpenTM program. Non-transactional code should behave exactly as described in the OpenMP specification. We discuss how the new OpenTM features interact with the existing OpenMP features in Section 3.3.5.

3.3.1 OpenTM Transactional Model

The transactional model of OpenTM is based on the following concepts.

Implicit Transactions: OpenTM supports implicit transactions. With OpenTM, the programmer simply defines transaction boundaries without any additional annotations for accesses to shared data. Implicit transactions reduce the burden on programmers but require compiler or hardware support to implement transactional bookkeeping. For STM systems, the compiler inserts the necessary TM barriers. For HTM systems, the hardware performs bookkeeping in background as the transaction performs regular memory operations [5]. Another advantage of implicit transactions is that they simplify software reuse. With implicit transactions, programmers can reuse a function without having to reconsider the TM barriers necessary for transactional bookkeeping under the new context.

Virtualized Transactions: OpenTM requires that the underlying TM system supports *virtualized transactions* that are not bounded by their execution time, memory footprint, and nesting depth. The underlying TM system should guarantee correct execution even when transactions exceed a scheduling quantum, exceed the capacity of hardware caches, or include a large number of nesting levels. Programmers will naturally expect that large transactions (even if less frequent than small transactions [33]) are correctly handled in a transparent manner. Virtualization is challenging for HTM systems because they use hardware caches for transactional bookkeeping.

Excluding the performance implications, OpenTM is agnostic to the exact method used for supporting virtualized transactions [10, 31, 32].

3.3.2 Basic OpenTM Constructs

This section presents the basic OpenTM constructs for parallel programming with transactions.

Transaction Boundaries: OpenTM introduces the `transaction` construct to specify the boundaries of memory transactions. The syntax is:

`#pragma omp transaction` *[clause[[,] clause]...]*
structured-block

where *clause* is one of the following: `ordered`, `nesting(open|closed)`. The `ordered` clause is used to specify a sequential commit order among concurrent transactions. This is useful for speculative parallelization of sequential code. If unspecified, OpenTM generates unordered yet serializable transactions. Unordered transactions are used for non-blocking synchronization in parallel code. During the execution of transactions, the underlying TM system detects conflicting accesses to shared variables to guarantee atomicity and isolation. On conflict, the system aborts and restarts transactions based on the ordering scheme and a contention management policy. We discuss the `nesting` clause along with the other advanced OpenTM features in Section 3.3.3. Note that the definition of *structured-block* is the same as OpenMP.

Transactional Loop: The `transfor` construct specifies a loop with iterations that concurrently execute as atomic transactions. The `transfor` syntax is:

`#pragma omp transfor` *[clause[[,] clause]...]*
for-loop

The `transfor` construct reuses most of the clauses of the OpenMP `for` construct such as `private` and `reduction` to identify private or reduction variables, respectively. A few clauses are extended or added to specify the transactional characteristics of the associated loop body. The `ordered` clause specifies that transactions will

commit in sequential order, implying a loop with sequential semantics (i.e., speculative loop parallelization). If `ordered` is not specified, `transfor` generates unordered transactions, which implies an unordered loop with potential dependencies.

The `transfor` construct can have up to three parameters in the `schedule` clause. Similar to the OpenMP `for` construct, the first parameter specifies how loop iterations are scheduled across concurrent worker threads. The second parameter specifies the number of iterations (*chunk size*), assigned to each thread on every scheduling decision. The tradeoff for the chunk size is between scheduling overhead and load balance across threads. The third parameter specifies how many loop iterations are executed per transaction by each worker thread (*transaction size*). If unspecified, a chunk of iterations is executed as a single transaction. The tradeoff for transaction size is between the overhead of starting and committing a transaction and the higher probability of conflicts among large transactions.

The following code example uses the `transfor` construct to perform parallel histogram updates. In this case, iterations are statically scheduled across threads with a chunk size of 7 iterations. Chunks are executed as unordered transactions.

```
void histogram(int *A,int *bin){
 #pragma omp transfor schedule(static,7)
 for(i=0;i<NUM_DATA;i++){
  bin[A[i]]++;
 }
}
```

The user can also define transactions using the `transaction` construct within the loop body of an OpenMP `for` construct. This approach allows programmers to tune the code with transactions smaller than a loop body, which may reduce the pressure on the underlying TM system. In contrast, it requires better understanding of the dependencies within the loop body and more coding efforts.

Transactional Sections: OpenTM supports transactions in parallel sections (i.e., non-iterative parallel tasks) using the `transsections` construct. Its syntax is:

`#pragma omp transsections` *[clause[[,] clause]...]*

[`#pragma omp transsection`*]*
structured-block 1
[`#pragma omp transsection`*]*
structured-block 2
...

Compared to OpenMP `sections`, `transsections` uses an additional `ordered` clause to specify the sequential commit order among concurrent transactions. While `sections` implies that the code blocks are proven to be parallel and independent, `transsections` can express parallel tasks with potential dependencies. Similar to the loop case, transactional sections can be also specified using the `transaction` construct within the OpenMP `sections`.

The following is a simple code example of method-level speculative parallelization using the OpenTM `transsections` construct:

```
#pragma omp transsections ordered {
  #pragma omp transsection
    WORK_A();
  #pragma omp transsection
    WORK_B();
  #pragma omp transsection
    WORK_C();
}
```

3.3.3 Advanced OpenTM Constructs

The basic OpenTM constructs discussed so far are sufficient to express the transactional parallelism in a wide range of applications. To support more advanced TM programming techniques, however, OpenTM introduces the following constructs. These constructs require support for advanced features in the underlying TM system.

Alternative execution paths: The `orelse` construct supports the alternative execution paths for aborted transactions [5, 49]. The syntax is:

```
#pragma omp transaction
  structured-block 1
#pragma omp orelse
  structured-block 2
...
```

If the transaction for the block 1 successfully commits, the entire code completes and the code in the block 2 does not executes. If the transaction executing the code in the block 1 aborts for any reason, the code associated with the `orelse` construct is executed as a transaction. Any arbitrary number of `orelse` constructs can be cascaded to specify alternative execution paths.

Conditional synchronization: OpenTM supports conditional synchronization in transactions using the `omp_retry()` runtime routine. `omp_retry()` indicates that the transaction is blocked due to certain conditions [49]. The runtime system decides whether the transaction is immediately re-executed or the corresponding thread will be suspended for a while. The transaction can use the `omp_watch()` routine to notify the runtime system that it should monitor a set of addresses and re-execute the blocked transaction when any of them has been updated [28]. The following is a simple code example of using conditional synchronization within a transaction:

```
#pragma omp transaction {
  if (queue.status == EMPTY) {
    omp_watch(addr);
    omp_retry();
  } else {
    t = dequeue(queue);}}
```

Compared to the conditional or guarded atomic blocks or condition variables [5, 48], conditional synchronization with *retry* allows for sophisticated blocking conditions that may occur anywhere within the transaction. Also, *retry* is composable [49]. If `omp_retry()` is called in a transaction that also uses `orelse`, the transaction is aborted and the control is immediately transferred to the alternative execution path.

Nested Transactions: The `nesting` clause specifies the behavior of nested transactions. If unspecified, OpenTM uses closed-nested transactions by default. Closed-nested transactions abort due to the dependencies without causing their parent to abort [66]. The memory updates made by closed-nested transactions become visible to other transactions only when the outermost transaction commits. The `open` clause allows a program to start an open-nested transaction that can abort independently from its parent but makes its updates visible to other transactions immediately upon its commit, even before the parent transaction [66] commits. Open-nested transactions may require finalizing and compensating actions performed when the parent transaction commits or aborts [47]. While we do not expect that non-expert programmers will directly use open-nested transactions, they can be helpful for addressing performance issues and implementing new language features [27, 28, 66].

Transaction Handlers: The `handler` construct specifies transactional handlers that are invoked when a transaction commits or aborts. The OpenTM handlers follow the semantics discussed in [66]. The handler syntax is:

`#pragma omp transaction` *[clause[[,] clause]...]*
  structured-block 1
`#pragma omp handler` *clause*
  structured-block 2

where clause is one of the following: `commit`, `abort`. `violation`. The `handler` construct can be associated with `transaction`, `transfor`, `transsections`, and `orelse` constructs. The code below illustrates an example with an abort handler used to compensate for an open-nested transaction:

```
#pragma omp transaction {
  #pragma omp transaction nesting(open)
  {
    WORK_A();
  } //end of open-nested transaction
  #pragma omp handler abort
  {
```

Runtime Routine	Description
`omp_in_transaction()`	Return true if executed within transaction.
`omp_get_nestinglevel()`	Return the nesting-level of the current transaction.
`omp_abort()`	User-initiated abort of the current transaction.
`omp_retry()`	User-initiated retry of the current transaction.
`omp_watch()`	Add an address to a watch-set [28].
`omp_set_cm()`	Set the contention management scheme.
`omp_get_cm()`	Return the current contention management scheme.

Table 3.1: The user-level runtime routines in OpenTM.

```
  COMPENSATE_WORK_A();
}} //end of parent transaction
```

3.3.4 Runtime System

OpenTM extends the runtime system of OpenMP to support transactional execution. Table 3.1 summarizes the user-level runtime library routines.

Loop Scheduling: The OpenMP `for` construct provides four options for scheduling iterations across worker threads: `static`, `dynamic`, `guided`, and `runtime`. Static scheduling statically distributes work to workers, while dynamic scheduling assigns a chunk of iterations to workers upon request at runtime. Guided scheduling is similar to dynamic scheduling, but the chunk size dynamically decreases to avoid work imbalance. Runtime scheduling makes a decision depending on the *run-sched-var* environment variable at runtime [2]. OpenTM reuses these options but adds an additional task of grouping loop iterations into transactions. With the `guided` option, the system can dynamically adjust the execution based on runtime feedback to reduce the runtime overheads of transactions by avoiding the virtualization overhead or frequent conflicts.

Contention Management: OpenTM provides two runtime routines, as shown in Table 3.1, to specify the contention management scheme of the underlying TM system [45, 84]. The `omp_get_cm()` routine returns the type of the currently used

contention management scheme. The `omp_set_cm()` routine allows the user to change contention management scheme for the whole system in runtime. Programmers can use this interface to adapt contention management to improve performance or provide fairness guarantees [45]. The exact parameters for `omp_set_cm()` depend on the available policies. For instance, our current implementation supports a simple back-off scheme [84] that requires a parameter to specify the maximum number of retries before an aborted transaction acquires priority to commit.

3.3.5 Open Issues and Discussion

There are a few open issues on OpenTM. Our current specification takes a conservative approach in several cases. However, we expect that extensive experiences with transactional applications and further developments in TM research will enable more sophisticated solutions.

OpenMP Synchronization: OpenTM disallows the use of OpenMP synchronization constructs within transactions (e.g., `critical`, `atomic`). OpenMP synchronization constructs have blocking semantics and can lead to various deadlock scenarios when used within transactions. OpenTM also disallows the use of atomic transactions within OpenMP synchronization constructs, as it can lead to deadlock scenarios if `omp_retry()` is used. In general, separating transactional code from blocking synchronization will help programmers reason about the correctness of their code. In addition, the blocking semantics of `atomic` and `critical` are the primary reason why we introduced the non-blocking `transaction` construct in OpenTM. Reusing `atomic` or `critical` for TM programming could lead to deadlocks or incorrect results for existing OpenMP programs.

I/O and System Calls: OpenMP requires that any library routines called within parallel regions are *thread-safe*. Similarly, OpenTM requires that any library routines called within transactions are *transaction-safe*. The challenge for TM systems is routines with permanent side effects such as I/O and system calls. At the moment, OpenTM does not propose any specification for such routines. Each OpenTM implementation is responsible for specifying which library calls can be safely invoked

within transactions as well as what are the commit and abort semantics. There is related research on how to support I/O and system calls within transactions using buffering and serialization techniques [66].

Relaxed Conflict Detection: A few papers have proposed directives that exclude certain variables from conflict detection (e.g., `race` [97] or `exclude` [68]). The key motivation is to reduce the TM bookkeeping overhead and to avoid any unnecessary conflicts. At the moment, OpenTM does not include such directives for the following reasons. First, without thorough understanding of the dependencies in the algorithm, using such directives can easily lead to incorrect or unpredictable program behavior. Second, the `private` and `shared` clauses in OpenMP already provide programmers with a mechanism to identify variables that are thread-private or participate in conflict detection between concurrent transactions.

Compilation Issues: For software TM systems, we should guarantee that any function invoked within a transaction is instrumented for TM bookkeeping. With function pointers or partial compilation, the compiler cannot statically identify such functions. In this case, programmers must use the `tm_function` directive to identify functions that may be called within transactions [99]. The compiler is responsible for cloning the function in order to insert TM barriers and the code necessary to steer control to the proper clone at any point in time.

3.4 OpenTM Implementation

Figure 3.1 summarizes the code generation process with the current OpenTM implementation. The application code is written in the C programming language with OpenTM directives. Note that programmers do not need to modify the application code to target a different TM system as command-line options can be used to guide the compiler to produce the code for the target TM system.

The current OpenTM implementation is based on a source-to-source OpenMP environment [95] and the transactional memory support in the GCC compiler [4].

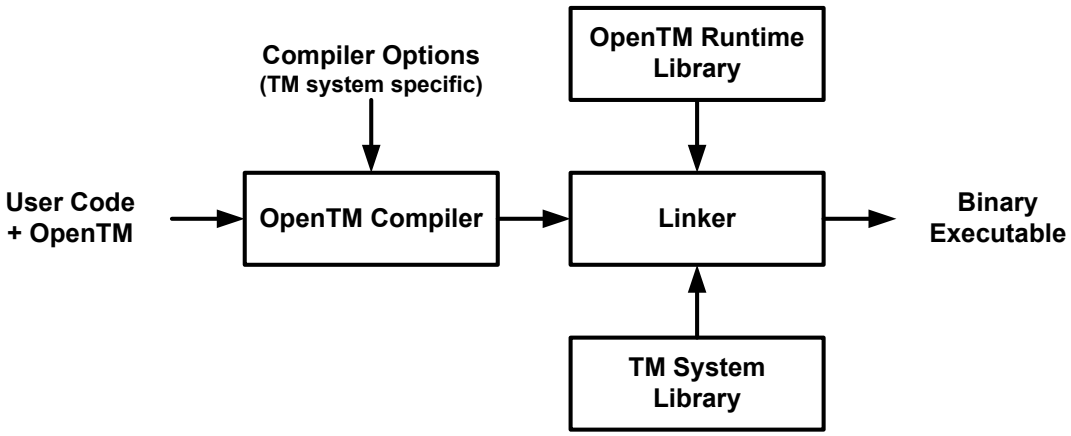


Figure 3.1: An overview of the current OpenTM implementation.

Specifically, the OMPi environment [95] is extended to perform high-level code translation in the C programming language to process OpenTM directives. The underlying GCC compiler then generates the code for hardware and software TM systems. While the compiler targets a single low-level TM interface shown in Table 3.2, it may use only a subset of the interface when targeting a specific TM system. For instance, hardware TM systems do not require the use of read and write barriers such as `TM_OpenWordForRead()` and `TM_OpenWordForWrite()`. The compiled code is linked with the OpenTM runtime library and the TM system library that provides the implementation of the low-level TM interface.

Low-level TM Interface: Table 3.2 shows the low-level TM interface targeted by the compiler. The interface describes the basic functionality in hardware and software TM systems. Columns H and S specify which functions are required by hardware and software TM systems, respectively. While these functions may be implemented differently across TM systems (e.g., software TM systems with eager [83] and lazy [37] version management), they are sufficient to control the execution of atomic transactions. Given an implementation of this interface, the compiler can generate correct code for any TM system.

The first set of functions in Table 3.2 are used to begin, commit, and abort transactions. These are common across all types of TM systems. OpenTM also requires mechanisms to register and invoke software handlers on abort or commit [66].

Low-level TM Interface	Description	H	S
`void TM_BeginClosed(txDesc*)`	Make a checkpoint and start a closed-nested transaction.	√	√
`void TM_CommitClosed(txDesc*)`	Commit a closed-nested transaction.	√	√
`void TM_BeginOpen(txDesc*)`	Make a checkpoint and start an open-nested transaction.	√	√
`void TM_CommitOpen(txDesc*)`	Commit an open-nested transaction.	√	√
`bool TM_Validate(txDesc*)`	Validate the current transaction.	√	√
`void TM_Abort(txDesc*)`	Abort the current transaction.	√	√
`void TM_RegHd(txDesc*,type, callbackFn*,params)`	Register a software handler; `txDesc`: transaction descriptor, `type`: commit, violation, abort, `callbackFn`: function pointer to the handler.	√	√
`void TM_InvokeHd(txDesc*,type, callbackFn*,params)`	Invoke a software handler.	√	√
`void TM_SetCM(cmDesc)`	Set the contention management policy; `cmDesc`: contention management descriptor (type & parameters).	√	√
`cmDesc TM_GetCM()`	Return a contention management descriptor for the current policy.	√	√
`uint32 TM_MonolithicReadWord (txDesc*, addr)`	Monolithic transactional read barrier.		√
`void TM_MonolithicWriteWord (txDesc*, addr, data)`	Monolithic transactional write barrier.		√
`void TM_OpenWordForRead (txDesc*, addr)`	Insert address in the transaction's read-set for conflict detection (decomposed read).		√
`uint32* TM_OpenWordForWrite (txDesc*, addr, data)`	Eager systems: insert into transaction's write-set and create undo log entry; Lazy systems: insert into transaction's write-set, allocate write-buffer entry and return its address.		√
`uint32 TM_ReadWordFromWB (txDesc*, addr)`	Lazy systems only: search write-buffer for address; if found, read value from write-buffer; otherwise return value in a regular memory location. Used when it is not certain at compile time, if a word has been written by this transaction or not.		√
`bool TM_ValidateWord (txDesc*, addr)`	Check the address for conflicts.		√

Table 3.2: The low-level TM interface targeted by the compiler.

Feature	Description
Processors	1–16 x86 cores, in-order, single-issue
L1 Cache	64-KB, 32-byte line, private 4-way associative, 1 cycle latency
Network	256-bit bus, split transactions pipelined, MESI protocol
L2 Cache	8-MB, 32-byte line, shared 8-way associative, 10 cycle latency
Main Memory	100 cycles latency up to 8 outstanding transfers

Table 3.3: Parameters for the simulated CMP system.

The second set of functions are the read and write barriers necessary for transactional bookkeeping in software TM systems. We support both monolithic barriers (e.g., `TM_MonolithicReadWord()`) and decomposed barriers (e.g., `TM_OpenWordForRead()` and `TM_ReadWordFromWB()` or `TM_ValidateWord()`). While monolithic barriers are sufficient for correct execution, decomposed barriers reveal more opportunities for performance optimizations [5].

Runtime System: The runtime system in the current OpenTM implementation extends the OpenMP runtime system with a small set of runtime library routines in Table 3.1. The current implementation supports dynamic scheduling for transactional loops. It also provides basic support for contention management. Conditional synchronization is currently implemented with immediate retries. It is interesting future work to integrate the runtime system with online profiling tools [30] in order to improve scheduling efficiency.

3.5 Experimental Methodology

3.5.1 Environment

We use an execution-driven simulator that models multi-core systems with MESI coherence and support for hardware TM systems. Table 3.3 summarizes the parameters for the simulated CMP architecture. All operations, except loads and stores, have

Application	Length	R/W Set Size	Time in Txn.
genome	Medium	Medium	High
intruder	Short	Medium	Medium
kmeans	Short	Small	Low
labyrinth	Long	Large	High
ssca2	Short	Small	Low
vacation	Medium	Medium	High
yada	Long	Large	High

Table 3.4: Summary of the characteristics of STAMP applications: length (the number of instructions), R/W set size, and the portion of the execution time spent in transactions [25].

a CPI of 1.0. However, all the details in the memory hierarchy timings, including contention and queueing events, are modeled. For the hardware TM system (HTM), the caches are enhanced with meta-data bits to support lazy version management and optimistic conflict detection [67]. We also use a eager-versioning TL2 software TM (STM) system [25, 37], running on top of a conventional multi-core system without hardware support for TM.

3.5.2 Applications

We used seven STAMP benchmarks [25] in our evaluation. Table 3.4 summarizes the characteristics of STAMP applications as reported in [25]. We also used a microbenchmark, histogram, in which multiple threads concurrently update an array of histogram bins after some computational work. The OpenTM code for all the applications was implemented using coarse-grain transactions to execute concurrent tasks that operate on concurrent data structures such as graphs and trees. Parallel programming at this level is straightforward because programmers do not have to manually manage the inter-thread dependencies within the data structure code. The parallel code is very close to the sequential code. For comparison, an additional version of the application code is used. This version directly used the low-level TM interface to implement another transactional version of the code with similar parallelization. Due to the

```
void client_run(args) {
  for(i=0; i<numOps; i++) {
    ...
    switch(action) {
      case MAKE_RESERVATION: {
        ...
        #pragma omp transaction
        { /* begin transaction */
          ...
          do_reservation();
          ...
        } /* end transaction */
        ...
      }
      ...
    }}}

void main() {
  #pragma omp parallel {
    client_run(args);
  }
}
```

Figure 3.2: OpenTM pseudocode for vacation.

lower-level syntax, this version tends to be cumbersome to develop but can provide some opportunities for optimizations that are not exposed at the OpenTM level.

3.5.3 Example: Vacation Pseudocode in OpenTM

To show the programming simplicity of OpenTM, Figure 3.2 presents the pseudocode of vacation. The programmer simply uses the `transaction` construct to specify that the client request code should run as atomic transactions. Unlike programming with low-level TM interface, OpenTM programming does not require manual instrumentation of shared memory accesses in transactions with TM barriers. In contrast with lock-based programming, there is no need to prove that the requests are independent or use low-level locks to manage infrequent dependencies. The OpenTM code

Data Structure	File	# of annotated C lines	
		LTM	OTM
heap	lib/heap.c	29	2
	lib/heap.h	0	2
list	lib/list.c	16	3
	lib/list.h	0	10
queue	lib/queue.c	23	0
	lib/queue.h	0	3
rbtree	lib/rbtree.c	102	14
	lib/rbtree.h	0	5

Table 3.5: The number of the annotated lines of C code needed to implement the transactional data structures in the STAMP benchmark suite using the low-level TM interface and OpenTM for STM systems.

achieves scalable performance because the underlying TM system implements optimistic concurrency and conflicts between client requests are typically infrequent. Using coarse-grain locks requires similar programming efforts, but does not scale due to serialization. Better scalability could be achieved using fine-grain locks, but it significantly increases programming complexity. The programmer must carefully manage the fine-grain locks as trees are traversed, modified, and rebalanced to avoid any possible deadlock scenarios.

3.6 Evaluation

3.6.1 Programmability

Quantifying programming efforts is a difficult task that requires extensive user studies. As an indication of coding complexity, Table 3.5 shows the number of the annotated C code lines to implement the transactional data structures in the STAMP benchmark suite using the low-level TM interface and OpenTM constructs for STM systems.

OpenTM has advantages over directly using a low-level TM (LTM) interface. For STM systems, using the low-level TM interface requires manual instrumentation and

optimizations of load and store operations within transactions. As it is most obvious with the case of rbtree (Table 3.5), the LTM code leads to significant code size increase and can be tricky to handle. If the programmer inserts redundant barriers, performance can be significantly degraded [5]. If there are missing barriers, the program behavior can be incorrect or unpredictable. OpenTM eliminates this tradeoff by automatically inserting and optimizing barriers for shared variables at compile time.

It should be also noted that the OpenTM approach can be advantageous even when targeting HTM systems. While read and write barriers are not needed in this case, significant code modification is often required to distribute and manage workloads across concurrent threads, which is highly error prone and complicates the optimizing process.

In line with OpenMP, the OpenTM code requires simple and high-level annotations for parallelism and memory transactions. It is straightforward to understand, maintain, and reuse the OpenTM code without requiring expert programmers or error-prone programming techniques.

3.6.2 Performance Comparison

Figures 3.3 and 3.4 show the speedups for the two code versions of STAMP applications, as we scale the number of processors from 1 to 16. Speedup is relative to the sequential version of each application (higher is better). Figure 3.3 and Figure 3.4 show the results on hardware and software TMs, respectively.

Figure 3.3 shows that the OpenTM code performs as well as the low-level TM interface (LTM) code on the HTM system. Since both versions utilize the hardware support for TM, they perform similarly for all system scales. While significantly improving the programmability with the simple and higher-level syntax, OpenTM does not cause any performance issues when applications execute on the HTM system.

In contrast, Figure 3.4 shows that the LTM code outperforms the OpenTM code for most of STAMP applications on the STM system. To better understand the performance differences, Figure 3.5 presents the execution times with 1 processor, normalized to the execution time of the LTM code. Execution time is broken into “busy”

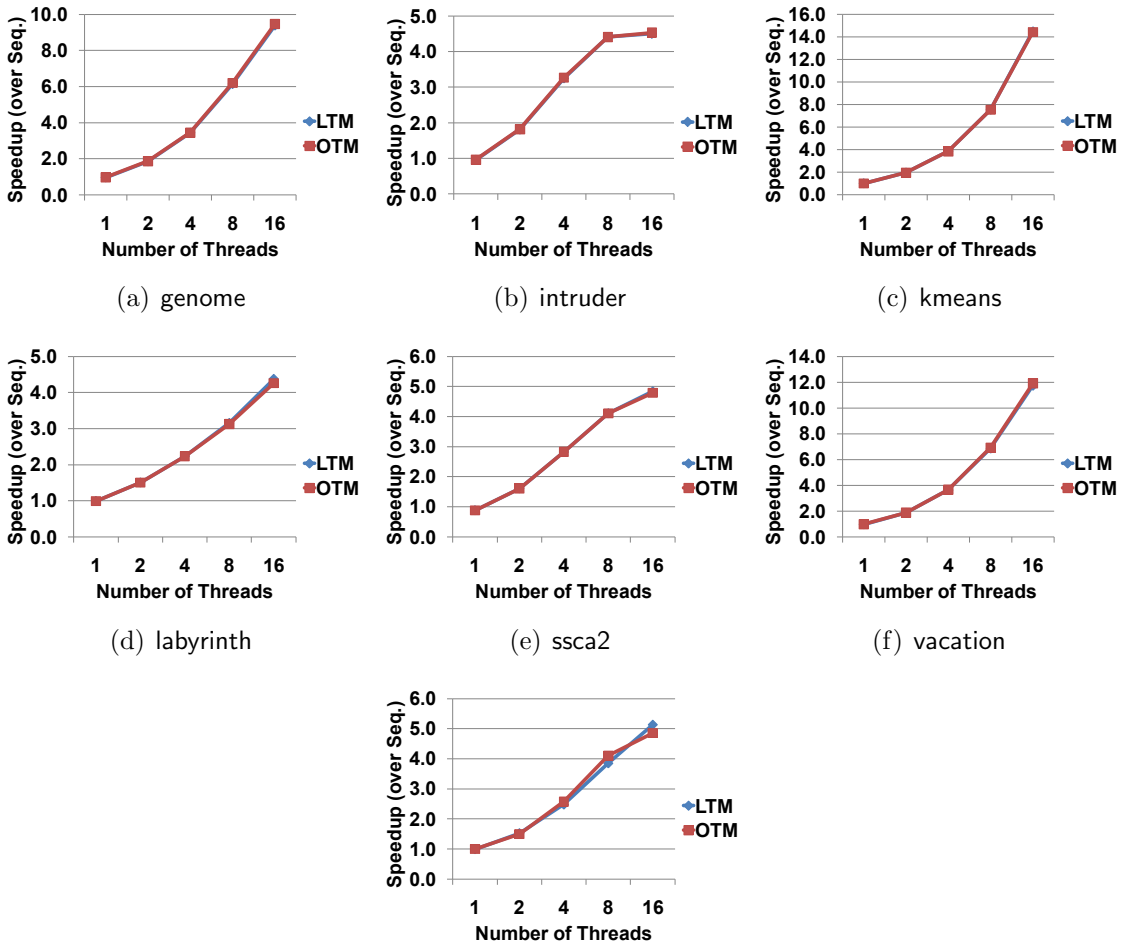


Figure 3.3: Application speedups with the low-level TM interface (LTM) and OpenTM (OTM) on an hardware TM.

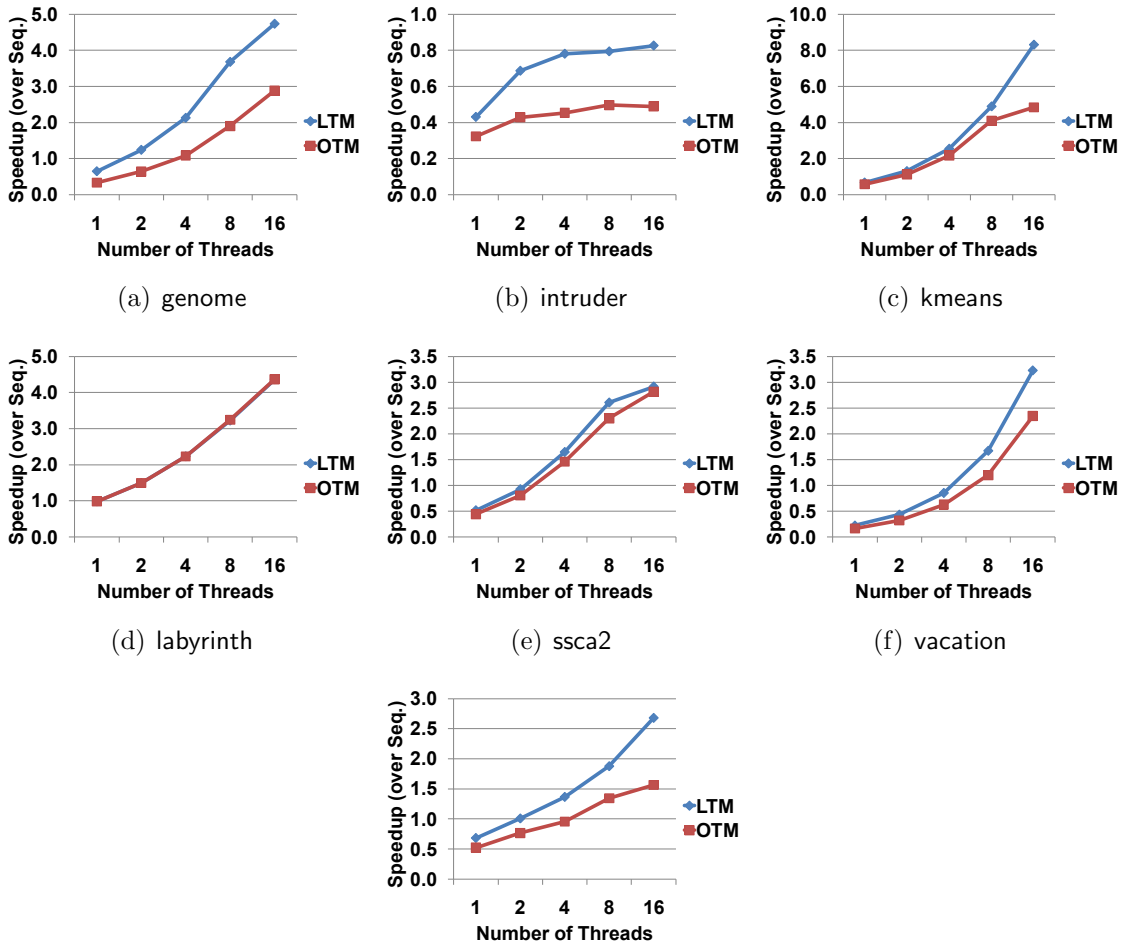


Figure 3.4: Application speedups with the low-level TM interface (LTM) and OpenTM (OTM) on a software TM.

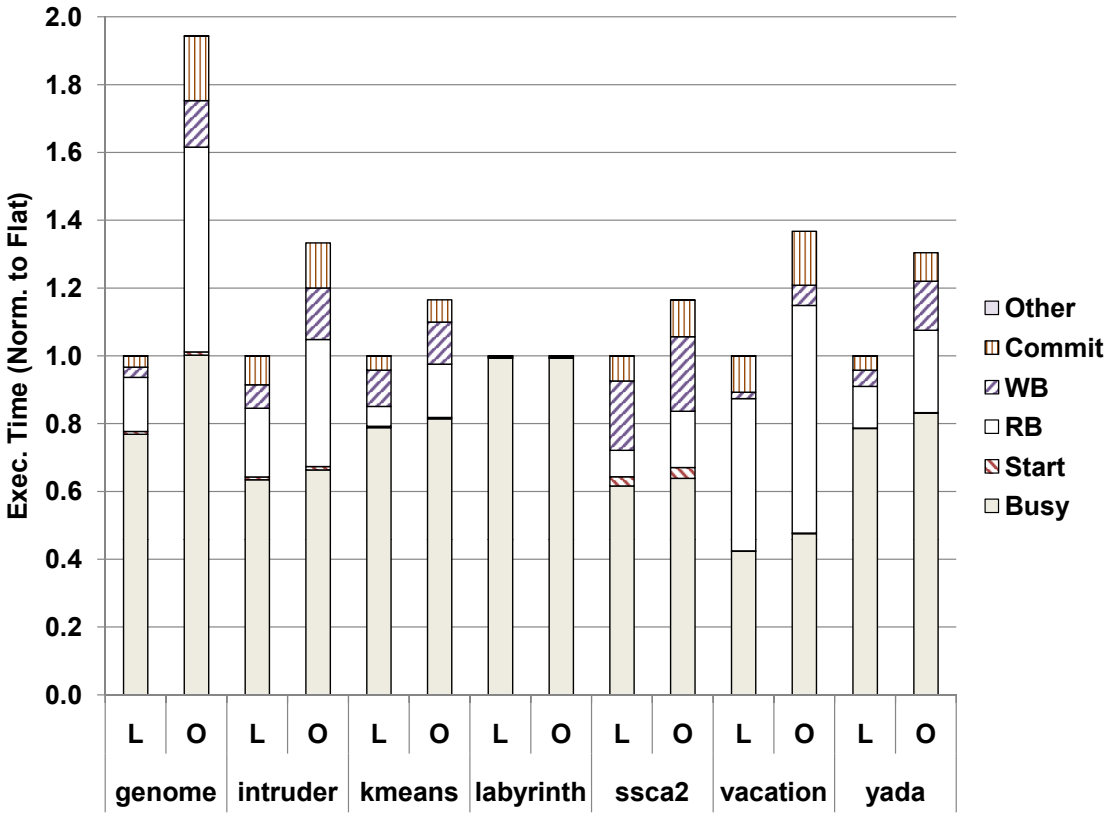


Figure 3.5: Normalized execution time with 1 processor. L and O indicate the results with the low-level TM interface and OpenTM, respectively.

(useful instructions and cache misses), "start" (the overhead of starting transactions), "RB" (read barriers), "WB" (write barriers), "commit" (the overhead of committing transactions), and "other" (work imbalance, etc.).

When an application spends a significant portion of the execution time (Table 3.4) in transactions and pointer variables are heavily used within transactions (e.g., genome, intruder, vacation, yada), the OpenTM code is significantly outperformed by the LTM code. For such applications, due to the limitations of static compiler analysis on pointer variables, the compiler must conservatively annotate memory accesses within transactions with TM barriers to ensure correctness. As a result, the over-instrumentation of shared accesses with TM barriers severely degrades the performance of the OpenTM code. In contrast, with the low-level TM

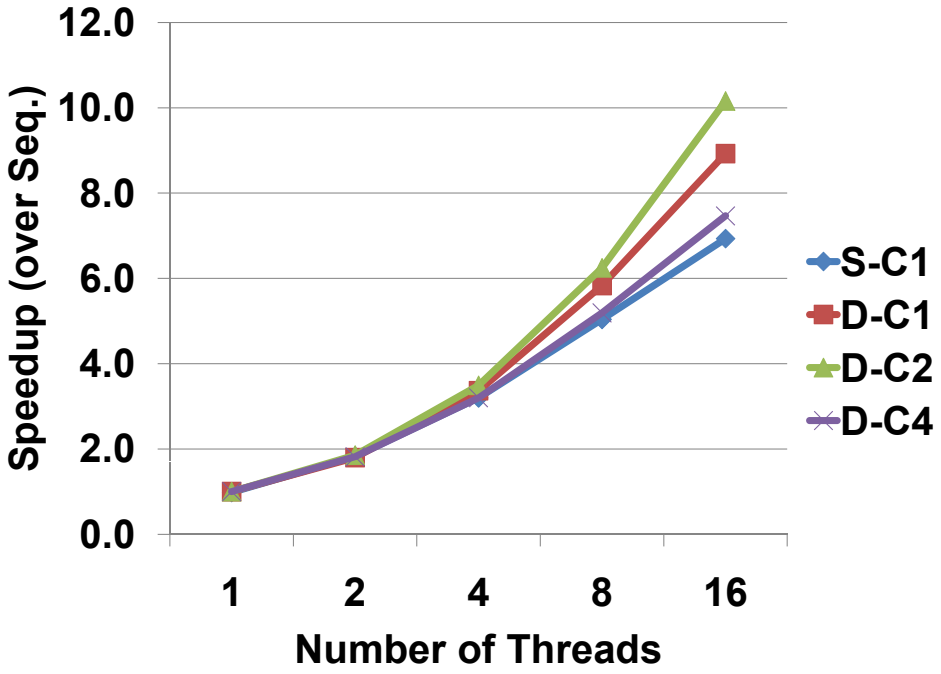


Figure 3.6: Speedups of the histogram microbenchmark with 16 processors and static (S) or dynamic (D) scheduling. C indicates the chunk size.

interface, programmers can avoid the over-instrumentation problem using application-level knowledge. For example, if a pointer variable used in transactions is guaranteed to be immutable based on high-level program semantics, it is safe and efficient to skip TM barriers for that variable. In this case, the LTM code can significantly outperform the OpenTM code by executing a smaller number of TM barriers. Finally, when an application spends a relatively small portion of the execution time (Table 3.4) in transactions (e.g., kmeans, ssca2) or executes functions that do not require TM cloning (e.g., labyrinth) within transactions, the OpenTM code performs comparably to the LTM code.

3.6.3 Runtime System

Dynamic Scheduling: To showcase the usefulness of the dynamic scheduling feature of OpenTM, we ran the histogram microbenchmark on the HTM system. Since

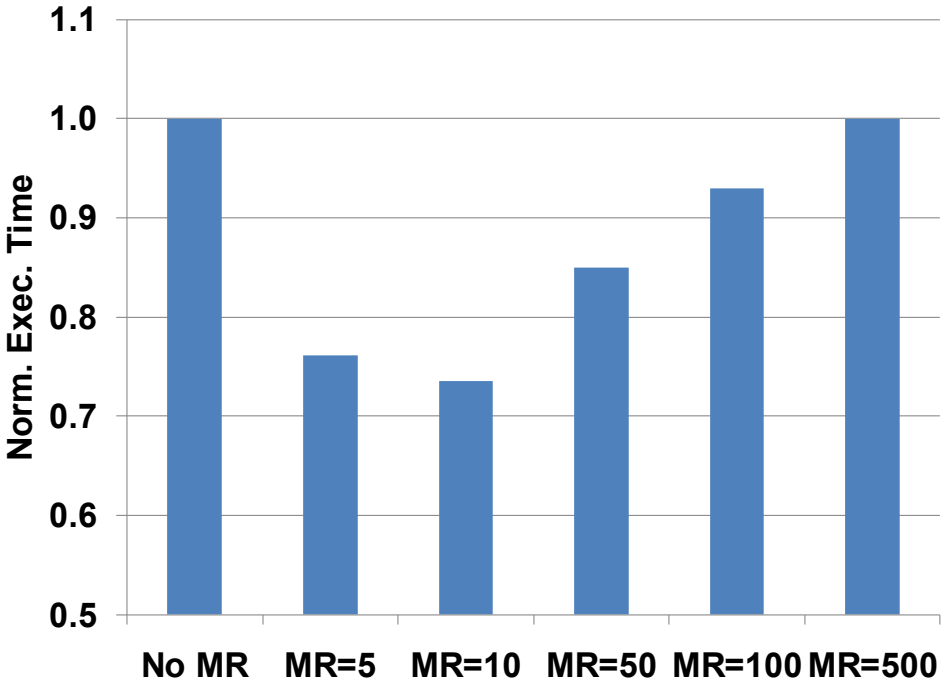


Figure 3.7: Normalized execution time for the histogram microbenchmark with 16 processors using various contention management configurations.

the microbenchmark performs a varying amount of computational work after each histogram bin is updated, there can be a significant work imbalance across threads. Figure 3.6 shows the speedups up to 16 processors. The *static* version uses the `static` clause to statically schedule loop iterations in an interleaved fashion with a chunk size of 1. The *dynamic* versions use the `dynamic` clause to dynamically schedule loop iterations with chunk size of 1 to 4 iterations. For simplicity, we set the transaction size equal to the chunk size. The dynamic versions scale better than the static version by mitigating work imbalance. The differences between the dynamic versions are due to the tradeoff between eliminating the scheduling overhead with smaller chunk size (e.g., frequently accessing the global work queue) and allowing for more frequent conflicts with a larger chunk size. Note that changing the scheduling policy and parameters is simple in OpenTM and does not require any intrusive code modifications, unlike the case with directly using the lower-level TM interface.

Contention Management: We also used the histogram microbenchmark to showcase the effectiveness of contention management in OpenTM. Figure 3.7 shows the normalized execution time with 16 processors and various contention management configurations. The leftmost bar shows the execution time with a linear backoff contention management scheme. The remaining bars show the execution time (normalized to the leftmost bar) with the following contention management scheme. When a transaction aborts, it uses linear backoff. If a transaction aborts *MR* times, it requests to become a high-priority transaction that wins the conflicts with other concurrent transactions. This priority mechanism provides fairness across threads, as large transactions do not starve due to the repeated conflicts with smaller transactions [22].

Figure 3.7 shows that the contention management policy with MR=10 reduces the execution time by 26%. It also shows the importance of tuning the contention management policy. If the MR value is low, the priority mechanism is frequently triggered, degrading performance due to serialization. On the HTM system we used, if a transaction triggers the priority mechanism, no other transactions are allowed to commit until the high-priority transaction commits, even if there is no potential conflict. If the MR value is high, performance also degrades because large transactions are repeatedly aborted by small transactions, causing imbalance in the system. Overall, contention management is important to achieve robust performance in TM systems. With OpenTM, users can provide high-level hints to guide the runtime system for more robust contention management.

3.7 Related Work

Several papers have proposed high-level programming environments for transactional memory or speculative parallel systems. Wang et al. proposed extensions for the C programming language and an optimizing compiler to support a software TM system [99]. Our proposal differs by building on top of a high-level programming environment (i.e., OpenMP). Instead of focusing only on software TM issues, we propose an integrated programming environment that includes TM features along with language

and runtime constructs to express and manage parallelism (e.g., work-sharing and dynamic scheduling). von Praun et al. proposed the IPOT programming environment to support speculative parallelization using ordered transactions [97]. OpenTM supports both ordered and unordered transactions, because we believe that non-blocking synchronization is as important as thread-level speculation. Moreover, IPOT lacks full-programming constructs such as work-sharing and advanced TM features that help programmers develop and optimize transactional applications.

In parallel with this work, Milovanovic et al. proposed to extend OpenMP to support TM [68]. Their proposal defines a transaction construct similar to the one in OpenTM and includes an implementation based on source-to-source translation. Apart from the transaction construct, however, OpenTM defines constructs for advanced TM features such as scheduling, conditional synchronization, nesting, transactional handlers, and contention management.

Several papers have proposed TM support for managed languages such as Java [5, 28, 49]. These proposals use similar language constructs for TM programming but exploit the features including dynamic code generation, type safety, and object-oriented of such languages. In contrast, OpenTM targets an unmanaged language environment (the C programming language), focusing on the synergy between high-level OpenMP programming and transactional memory techniques for non-blocking synchronization and speculative parallelization.

3.8 Conclusions

This chapter presented OpenTM, a high-level API for transactional programming. OpenTM extends OpenMP, a widely-used high-level parallel programming environment, to support both non-blocking synchronization and speculative parallelization using atomic transactions. We also presented a prototype OpenTM implementation based on a source-to-source OpenMP environment and the transactional memory support in the GCC compiler. The system can produce code for hardware and software TM systems. It also includes a runtime system with the features such as dynamic

scheduling and contention management. We evaluated the programmability and performance of the OpenTM environment and demonstrated that it delivers good performance with simple and portable high-level code. In summary, OpenTM provides a practical and efficient transactional programming environment within the familiar scope of OpenMP.

Chapter 4

Nested Parallelism in STM

4.1 Introduction

To date, most TM systems have assumed that the code inside a transaction executes sequentially. However, real-world applications often include the potential for *nested parallelism* in various forms such as nested parallel loops, calls to parallel libraries, and recursive function calls [91]. With the increasing number of cores in the system, it is important to fully exploit the parallelism available at all the levels to achieve the best possible performance. In this spirit, several parallel programming models with support for nested parallelism have been proposed [2, 90]. Hence, to maximize performance gain and integrate well into popular programming models, TM must support nested parallelism.

However, efficiently exploiting nested parallelism in TM is not trivial. The key challenge of nested parallelism is to amortize the runtime overheads for initiating, synchronizing, and balancing the inner-level, fine-grained parallelism [17]. Nested parallelism within transactions exacerbates this challenge due to the extra runtime overheads for initiating, versioning, and committing nested transactions. Designing a TM system that supports nested parallel transactions is also challenging. First, the conflict detection scheme must correctly track dependencies in a hierarchical way

instead of a flat manner. Nested parallel transactions may conflict and restart without necessarily aborting their parent transactions. Second, we must ensure that the memory overhead required for tracking the state of nested transactions is small.

A few recent papers on nested parallelism in STM have discussed the semantics of nested parallel transactions and presented prototype implementations [8, 15, 80, 96]. The following questions, however, still require further investigation. First, what is a cost-effective algorithm to enable nested parallelism in high-performance STMs? Second, based on a detailed performance analysis, what are the practical tradeoffs and issues when exploiting nested parallelism in STM? Answering these questions is also important to guide future research on nesting-aware TM runtime environments.

This chapter presents *NesTM*, an STM system with support for closed-nested parallel transactions. NesTM is based on a high-performance, blocking STM that uses eager data versioning and word-granularity conflict detection. NesTM extends the baseline STM to support nested parallel transactions in a way that keeps state overheads small.

The specific contributions of this work are:

- We propose an STM system (NesTM) that supports nested parallelism with transactions and parallel regions nested in an arbitrary manner.

- We discuss several complications of concurrent nesting, describe solutions for addressing the issues, and discuss their impact on performance.

- We quantify the performance of nested parallel transactions on NesTM when the available parallelism is exploited in deeper nesting levels.

The rest of this chapter[1] is organized as follows. Section 4.2 provides background information. Section 4.3 describes NesTM and Section 4.4 discusses subtle correctness issues. Section 4.5 presents the quantitative evaluation. Section 4.6 reviews related work. Finally, Section 4.7 concludes this chapter.

[1]The work presented in this chapter was also published in [12].

4.2 Background

Baseline STM

Our starting point is a timestamp-based blocking STM algorithm that uses eager data versioning [50, 83]. This approach has been shown to have performance advantages over non-blocking or lazy-versioning STMs and has been used by the Intel STM compiler [83] and the Microsoft Bartok environment [50]. While we focus on an STM with word-granularity conflict detection, our findings can apply to STMs with object-granularity conflict detection.

The baseline STM builds upon the following key data structures: (1) a *global version clock*, a shared variable used to establish serializability and (2) a set of variables (*voLocks*), each of which acts as a lock or stores a version number (i.e., the clock value when the memory object was written by a committing transaction) of the associated memory object. We refer to Section 2.4.2 for the additional details on the eager-versioning STM system.

Semantics of Concurrent Nesting

We describe a few key concepts for nested parallel transactions. Longer discussions of nested parallel transactions are available in [8, 72].

Definitions and concepts: At runtime, each transaction is assigned with a *transaction ID (TID)*, which is a unique positive integer. The *root* transaction (TID 0) is reserved to represent the globally-committed state of the system. Every non-root transaction has a unique *parent* transaction. Specifically, *top-level* transactions are the ones whose parent is the root transaction. Following the semantics in [72], we assume that a transaction does not perform transactional operations concurrently with any of its (live) descendants. Finally, $\mathsf{family}(T)$ of a transaction T is defined as a union of $\mathsf{ancestors}(T)$ and $\mathsf{descendants}(T)$.

Transactional semantics: We describe the definition of conflict described in [8] for TM systems with closed nesting. For a memory object l, let $\mathsf{readers}(l)$ be a set of

active transactions that have l in their read-sets. $\mathsf{writers}(l)$ is defined similarly. When a transaction T accesses l, the following two cases are conflicts:

- T reads from l: if there exists a transaction T' such that $T' \in \mathsf{writers}(l)$, $T' \neq T$ and $T' \notin \mathsf{ancestors}(T)$.
- T writes to l: if there exists a transaction T' such that $T' \in \mathsf{readers}(l) \cup \mathsf{writers}(l)$, $T' \neq T$ and $T' \notin \mathsf{ancestors}(T)$.

As for the commit semantics, if T is not a top-level transaction, its read- and write-sets are merged into those of its parent. Otherwise, all the values written by T become visible to the other transactions and its read- and write-sets are reset. If T aborts, all the updates made by T are discarded and the previous state is restored [72].

4.3 Design and Implementation of NesTM

This section describes the NesTM algorithm, an execution example, and the performance issues.

4.3.1 NesTM Algorithm

The key design goal of NesTM is to the keep state overhead small in supporting nested parallel transactions. For example, we do not want to significantly increase the memory footprint by using multiple sets of locks and global version clocks to support multiple nested parallel regions. The blocking, eager versioning STM used as our baseline STM has a useful property that helps us meet our goal: once a transaction writes (i.e., acquires a lock) to a memory object, it is guaranteed to have the exclusive ownership for the object until the transaction commits or aborts.

Before discussing the NesTM algorithm in detail, we describe the changes in the version-owner locks (voLocks) compared to those in the baseline STM. As shown in Figure 4.1, the voLock in the baseline STM is a word-sized data structure (e.g., N=32 on a 32-bit machine) that encodes the version number or the owner information on the associated memory object. If L=1 (i.e., locked), the remaining N-1 bits store

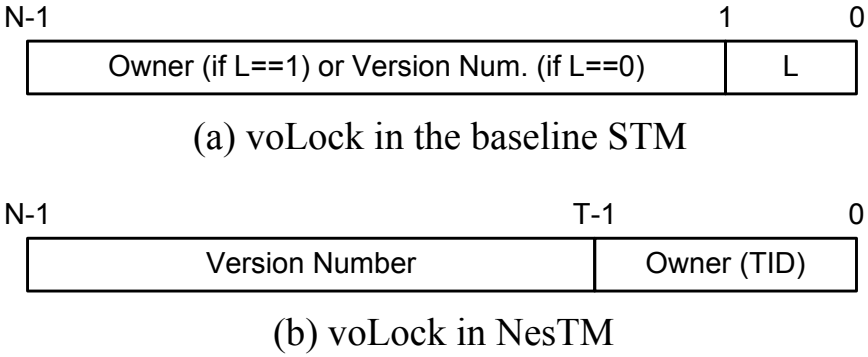


Figure 4.1: Comparison of voLocks.

the owner information. If L=0 (i.e., unlocked), the remaining N-1 bits store the version number. This encoding is sufficient for supporting only top-level transactions because if a transaction locks a memory object, no other transactions are allowed to access the object until the transaction commits or aborts. In NesTM, however, other transactions should be allowed to access the locked object as long as they are descendants of the owner. To allow this, the ownership information should be always available in the voLock to consult the ancestor relationship at any time. Similarly, the version number in the voLock should be always available to serialize conflicting transactions.

For this purpose, we modify the voLock as shown in Figure 4.1. The T least significant bits (LSBs) are used to encode the owner of the associated object. Since the TID 0 is reserved for the root transaction, NesTM can support up to $2^T - 1$ concurrent transactions. While we use T=10 (i.e., 1023 concurrent transactions) in this work, it is tunable. The remaining N-T bits in the voLock store the version number. Since the global version clock increments by 2^T when each transaction commits, it can saturate faster than the baseline STM. Recent work describes how to handle the version clock overflow [39].

Figures 4.2, 4.3, and 4.4 provide the pseudocode for the NesTM algorithm. We summarize its key functions below.

TxStart: This barrier is almost identical to the one in the baseline STM except that it returns "fail" if there is any doomed ancestor of the transaction we attempt

```
 1: procedure isInReadSet(Self, addr)
 2:   acquireLock(Self.commitLock)
 3:   result ← addr ∈ Self.RS
 4:   releaseLock(Self.commitLock)
 5:   return result

 6: procedure DoomHighestConflictTx(Self, Owner)
 7:   ptr ← Self
 8:   while ptr.Parent ∉ {Root,Owner,Ances(Owner)} do
 9:     ptr.doomed ← true
10:     ptr ← ptr.Parent

11: procedure TxStart(Self)
12:   Self.aborts ← 0
13:   checkpoint()
14:   if isAnyDoomedAnces(Self) then
15:     return fail
16:   Self.doomed ← false
17:   Self.rv ← GlobalClock
18:   return success

19: procedure TxLoad(Self, addr)
20:   if Self.doomed = true or isAnyDoomedAnces(Self) then
21:     TxAbort(Self)
22:   retry_load:
23:   rb ← RollbackCounter
24:   cv ← getVoVal(addr)
25:   Owner ← extractOwn(cv)
26:   value ← Memory[addr]
27:   if Owner = Self then
28:     Self.RS.insert(addr)
29:     return value
30:   else if Owner∈Ances(Self) and cv = getVoVal(addr) then
31:     if rb ≠ RollbackCounter then
32:       goto retry_load
33:     if extractTS(cv) > Self.rv then
34:       TxAbort(Self)
35:     else
36:       Self.RS.insert(addr)
37:       return value
38:   else
39:     if Owner∉Ances(Self) and Self.aborts%p = p − 1 then
40:       DoomHighestConflictTx(Self, Owner)
41:     TxAbort(Self)
```

Figure 4.2: Pseudocode for the basic functions in NesTM.

```
1:  procedure ValidateReaders(Self, Owner, addr)
2:    ptr ← Self
3:    hcr ← NIL
4:    while ptr ∉ {Root,Owner,Ances(Owner)} do
5:      if getTS(addr) > ptr.rv and isInReadSet(ptr, addr) then
6:        ptr.doomed ← true
7:        hcr ← ptr
8:      ptr ← ptr.Parent
9:    return hcr

10: procedure TxStore(Self, addr, data)
11:   if Self.doomed = true or isAnyDoomedAnces(Self) then
12:     TxAbort(Self)
13:   Owner ← getOwner(addr)
14:   if Owner = Self then
15:     cv ← getVoVal(addr)
16:     Self.WS.insert(addr, Memory[addr], cv)
17:     Memory[addr] ← data
18:   else
19:     cnt ← 1
20:     repeat
21:       cv ← getVoVal(addr)
22:       ov ← cv
23:       nv ← extractTS(cv) | Self.TID
24:       if extractOwner(cv) ∈ Ances(Self) then
25:         ov ← atomicCAS(getVoAddr(addr), cv, nv)
26:         if ov = cv then
27:           hcr←ValidateReaders(Self,extractOwner(cv),addr)
28:           if hcr ≠ NIL then
29:             setVoVal(addr, cv)
30:             TxAbort(Self)
31:           Self.WS.insert(addr, Memory[addr], cv)
32:           Memory[addr] ← data
33:           return
34:       cnt ← cnt + 1
35:     until cnt = C
36:     if Self.aborts % p = p − 1 then
37:       DoomHighestConflictTx(Self, extractOwner(ov))
38:     TxAbort(Self)
```

Figure 4.3: Pseudocode for the basic functions in NesTM.

```
 1: procedure TxCommit(Self)
 2:   if Self.doomed = true or isAnyDoomedAnces(Self) then
 3:     TxAbort(Self)
 4:   wv ← Fetch&Increment(GlobalClock)
 5:   acquireLock(Self.Parent.commitLock)
 6:   for all e in Self.RS do
 7:     cv ← getVoVal(e.addr)
 8:     Owner ← extractOwner(cv)
 9:     if Owner = Self then
10:       continue
11:     else if Owner ∈ Ances(Self) then
12:       if extractTS(cv) > Self.rv then
13:         releaseLock(Self.Parent.commitLock)
14:         TxAbort(Self)
15:     else
16:       releaseLock(Self.Parent.commitLock)
17:       if Self.aborts % p = p − 1 then
18:         DoomHighestConflictTx(Self, Owner)
19:       TxAbort(Self)
20:   mergeRWSetsToParent(Self)
21:   releaseLock(Self.Parent.commitLock)
22:   for all e in Self.WS do
23:     Owner ← getOwner(e.addr)
24:     if Owner = Self then
25:       nv ← wv | Self.Parent.TID
26:       setVoVal(e.addr, nv)
27:   Self.RS.reset()
28:   Self.WS.reset()

29: procedure TxAbort(Self)
30:   Self.doomed ← false
31:   Self.aborts ← Self.aborts + 1
32:   Self.RS.reset()
33:   atomicIncrementRollbackCounter()
34:   for all e in Self.WS do
35:     Memory[e.addr] ← Self.WS.lookup(e.addr)
36:   for all e in Self.WS do
37:     Owner ← getOwner(e.addr)
38:     if Owner = Self then
39:       setVoVal(e.addr, e.prevVoLock)
40:   Self.WS.reset()
41:   doContentionManagement()
42:   restoreCheckpoint()
```

Figure 4.4: Pseudocode for the basic functions in NesTM.

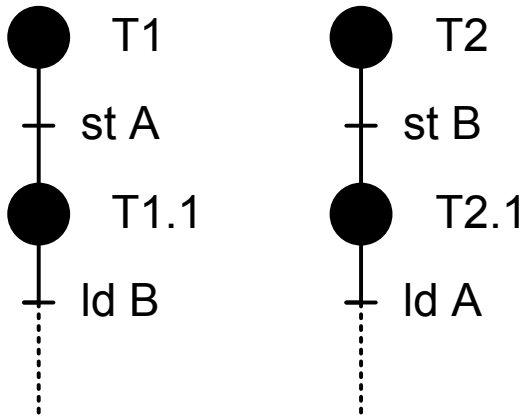


Figure 4.5: A livelock scenario avoided by eventual rollback of the outer transaction.

to start. The return value is used to initiate recursive aborts to provide forward progress.

TxLoad: Following the definitions of the nesting-aware conflicts discussed in Section 4.2, a transaction is allowed to read a memory object only if the owner of the object is itself or its ancestor. When the transaction is the owner, it can safely read the memory object without checking the version number (the reason will be explained in the discussion of TxStore). If the owner is its ancestor, it relies on the version number to establish serializability. If the owner is neither itself nor its ancestor, the transaction conflicts with the owner. In lines 39–40 in Figure 4.2, it periodically calls DoomHighestConflictTx. This is to avoid potential livelock scenarios. Figure 4.5 illustrates an example. If only nested transactions (i.e., T1.1 and T2.1) abort and restart, none of them can make forward progress as the memory objects are still (crosswise) locked by their ancestors. To avoid livelock, at least one of the ancestors should abort and release the locked memory objects. For this purpose, NesTM periodically checks and dooms ancestors. Note that we could use a more precise livelock detection mechanism, but it would also incur a significant runtime overhead. Also note that similar livelock scenarios exist even in the baseline STM. Finally, note that the *RollbackCounter* is used to avoid the *invalid-read* problem discussed in Section 4.4.1.

TxStore: When a transaction attempts to write to a memory object, it is allowed to do so if it is the owner of the memory object. Otherwise, it attempts to acquire the lock for the memory object, only if the owner is an ancestor. If it fails, the transaction conflicts and DoomHighestConflictTx is periodically called to avoid any potential livelock (lines 36–37 in Figure 4.3). If it successfully acquires the lock, it calls ValidateReaders with parameters such as Self, Owner (the previous owner for the object), and addr. In ValidateReaders, the transaction itself and all its ancestors that are also not an ancestor of Owner are validated for this memory object (lines 1–9 in Figure 4.3). The key insight here is that once a transaction T or any of its descendants writes (i.e., acquires the lock) to a memory object, T is guaranteed to have the exclusive ownership for the object until it commits or aborts. Therefore, if we ensure that there have been no conflicting writes to an object for T and all of its ancestors at the time when T first attempts to write to the object, the object is guaranteed to be valid throughout T and its ancestors' execution. If there is any invalid reader, it transfers the ownership to the previous owner and triggers rollback (lines 28–30 in Figure 4.3). Note that validating each transaction is protected by the *commit-lock* of the validated transaction to avoid the problem with the non-atomic commit discussed in Section 4.4.2. Also, note that TxStore can be expensive when a transaction executes in a deep nesting level due to the read-set search for itself and its ancestors. We will discuss this performance issue in Section 4.3.3.

TxCommit: If a transaction or any of its ancestors is doomed, it aborts (lines 2–3 in Figure 4.4). Otherwise, the transaction validates all the read-set entries (lines 6–19 in Figure 4.4). If the read-set is successfully validated, it merges its read- and write-sets to those of its parent (line 20 in Figure 4.4). Note that to avoid the problem with the non-atomic commit discussed in Section 4.4.2, the process of validating and merging the read-set is protected by the commit-lock of the parent. To reduce the execution time in the critical section, merging is done by linking (i.e., instead of copying) the pointers in read- and write-sets implemented using linked-lists. Then, the version number and ownership for each memory object in the write-set are incremented and transferred to the parent.

TxAbort: After updating the transactional metadata and incrementing the RollbackCounter, the write-set is traversed backward (i.e., from the newest to oldest) to roll back the speculatively-written memory values. Then, the write-set is traversed forward (i.e., from the oldest to newest) to restore the value of each voLock to the first observed value. Note that the voLock is released only if the owner of the memory location is the transaction itself (line 38 in Figure 4.4) to address the double-release problem [96]. Finally, the checkpoint is restored to restart the transaction.

Note that by calling DoomHighestConflictTx in TxLoad, TxStore, and TxCommit, possible livelock scenarios similar to the one shown in Figure 4.5 can be avoided. In addition, a randomized exponential backoff scheme is used for contention management.

4.3.2 Example

Figure 4.6 shows an example of how a simple application using nested parallel transactions runs on NesTM. Initially, GC=0 and TS(A)=TS(B)=0. Note that GC is incremented by 2^{10} in the actual implementation. For simplicity, we assume GC is incremented by 1 in the examples in this chapter.

At (wall clock) time 0, T1 starts (RV(T1)=0). At time 2, T1 reads B. At time 3 and 4, T2 starts (RV(T2)=0) and writes to A. At time 5, T2 commits and GC=1 and TS(A)=1. At time 6, threads executing T1.1 and T1.2 (children of T1) are forked and T1.1 and T1.2 start (RV(T1.1)=RV(T1.2)=1). At time 7, both T1.1 and T1.2 successfully read A because RV(T1.1)=RV(T1.2)$\geq$TS(A). At time 8, T1.2 attempts to write to A. T1.2 validates itself and its ancestors (T1) by calling ValidateReaders. T1.2 is valid because A is in its read-set and RV(T1.2)$\geq$TS(A). T1 is not doomed because A is not in its read-set (read-sets of T1.1 and T1.2 have not been merged yet). Hence, T1.2 can successfully write to A. At time 9, T1.2 successfully commits and GC=2 and TS(A)=2. In addition, the read- and write-sets of T1.2 are merged into the ones of T1. At time 10, T1.1 attempts to commit but fails because A is in the read-set of T1.1 and RV(T1.1)$<$TS(A). GC is incremented to 3 due to this unsuccessful commit.

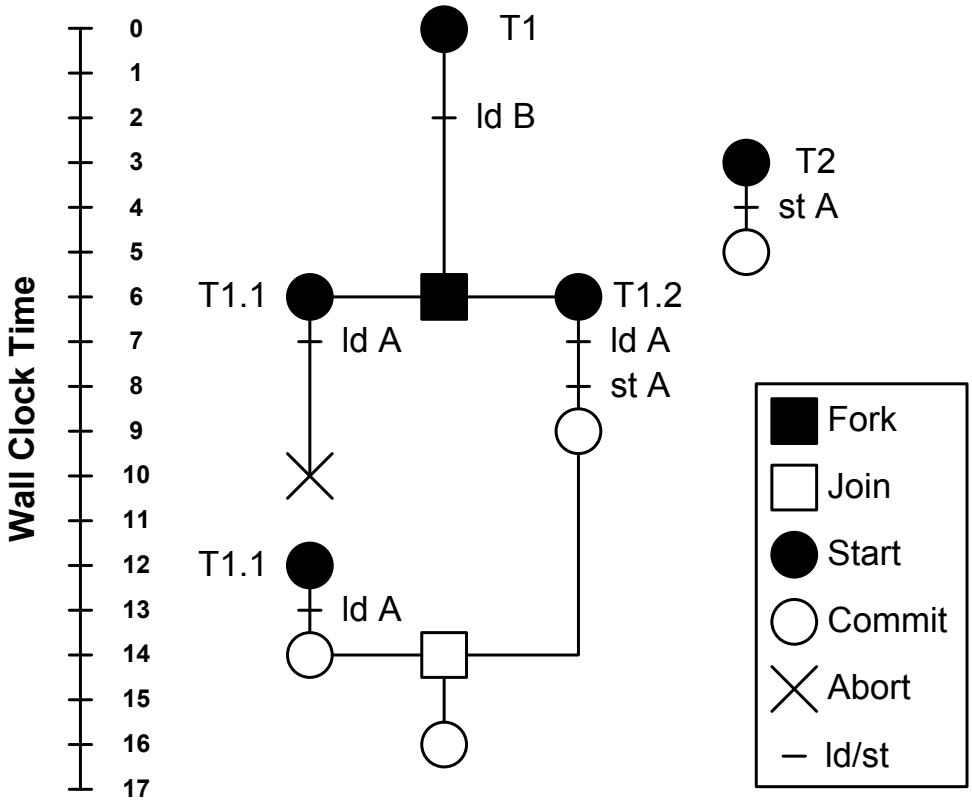


Figure 4.6: An example of a TM application running on NesTM.

At time 12, T1.1 restarts (RV(T1.1)=3). At time 13, T1.1 successfully reads A because the owner of A is T1 (an ancestor of T1.1) and RV(T1.1)≥TS(A). At time 14, T1.1 successfully commits, GC is incremented to 4 (but still TS(A)=2), and T1 resumes its execution after child threads join. At time 16, T1 successfully commits because it has an ownership for A (i.e., transferred from T1.2) and RV(T1)≥TS(B). Also, GC and TS(A) are incremented to 5.

4.3.3 Qualitative Performance Analysis

Table 4.1 summarizes a symbolic comparison of the common- and worst-case time complexity of TM barriers in baseline STM and NesTM. NesTM can be implemented in two ways: (1) NesTM-L: a baseline implementation that uses linked-lists to implement read- and write-sets and (2) NesTM-H: an advanced implementation that uses

	Baseline STM	NesTM-L (Linked list)		NesTM-H (Hash)	
		Common Case	Worst Case	Common Case	Worst Case
Read	$O(1)$	$\sim O(1)$	$O(d)$	$\sim O(1)$	$O(d)$
Write	$O(1)$	$\sim O(1)$	$O(d \cdot R)$	$\sim O(1)$	$O(d)$
Cmt.	$O(R+W)$	$\sim O(R+W)$	$O(d+R+W)$	$\sim O(R+W)$	$O(d+R+W)$

Table 4.1: A symbolic comparison of the common- and worst-case time complexity of TM barriers in the baseline STM and NesTM. R, W, and d indicate the read-set size, write-set size, and nesting depth, respectively.

hash tables to implement read- and write-sets. The current NesTM implementation follows NesTM-L. We assume the common case is the one in which the nesting depth is small and there is strong temporal locality between reads and writes (i.e., a transaction writes to a recently-read memory object). On the other hand, we assume that the worst case is an uncommon case in which the nesting depth is large and there is weak temporal locality between reads and writes.

In the common case, the time complexity of NesTM-L TM barriers can be almost similar to the ones in the baseline STM because the nesting depth is small (i.e., $d \simeq 1$) and only a few entries in the read-set need to be looked up at each write to check the validity due to the strong temporal locality between reads and writes. In the worst case, however, the time complexity of NesTM-L TM barriers is significantly higher than the baseline STM. On the other hand, NesTM-H still shows comparable time complexity as the baseline STM due to the use of hash tables.

Apart from the differences in the time complexity of TM barriers, there are additional performance issues to note. First, temporal locality is lost when accessing the transactional metadata of nested transactions. Since, when a child transaction commits, its read- and write-set entries are merged to its parent, there is no temporal locality for these entries when a new transaction begins on the same core. Second, the same memory objects in the read-set are repeatedly validated across different nesting levels. Finally, when a large number of child transactions simultaneously attempt to commit, contention on the commit-lock of the parent can become a critical performance bottleneck. We quantify these performance issues in Section 4.5.

4.4 Complications of Concurrent Nesting

We now discuss subtle correctness issues we have encountered while developing NesTM. We also describe our efforts on the correctness and liveness of NesTM.

4.4.1 Invalid Read

Problem: In the read barrier, reading a voLock and the associated memory value does not occur atomically. Because of this, eager STMs are potentially vulnerable to the *invalid-read* problem[2]. A transaction may incorrectly read an invalid memory value speculatively written by an aborting transaction. If the aborting transaction restores the original voLock value, the validation process at the end of the read barrier will miss the problem. In flat STMs, this problem can be avoided by always incrementing the timestamp values of voLocks even when an aborting transaction releases them. In NesTM, however, this technique cannot be used due to the *self-livelock* problem. If an aborting descendant increments the timestamp value of the voLock for a memory object, its ancestor that has the memory object in its read-set can be aborted due to the incremented timestamp value of the voLock. Consequently, the subtree rooted by the ancestor cannot make any forward progress.

Solution: To address both invalid-read and self-livelock problems at the same time, we propose the *RollbackCounter* scheme. On abort, a transaction atomically increases the global RollbackCounter in addition to restoring the values of the voLocks in its write-set to the first observed values. When a transaction attempts to read a memory object, it first samples the value of the RollbackCounter before reading the value of the associated voLock (line 23 in Figure 4.2). After ensuring that the voLock value remains unchanged (line 30), the previously sampled value of the RollbackCounter is compared with the current value. If the two values match, it is guaranteed that there has been no aborting transaction since the voLock value was read, thus no possibility of the invalid read. If the two values differ, the transaction conservatively avoids the invalid-read problem by retrying the whole process (line 32).

[2] Section 6.3.4 discusses the invalid-read problem in detail.

Performance impact: Since only a single, global RollbackCounter is used, false positives can degrade the performance by making transactions repeat the process several times even when they did not actually read invalid memory values. Furthermore, the extra code added to access the RollbackCounter in the read barrier can degrade the performance.

Possible alternatives: Instead of using the eager version management (VM) scheme, a lazy VM scheme could be used (while still using the encounter-lock scheme) to avoid the invalid-read problem. However, it can cause significant performance issues because the write-set of a transaction is frequently accessed by the transaction itself and its descendants.

4.4.2 Non-atomic Commit

Problem: Figure 4.7 shows a potential serializability violation scenario due to the non-atomic commit. Initially, GC and TS(A) are set to 0. After reading A at time 3, T1.1 initiates its commit at time 4. At time 5, A is validated. At time 6, T2 writes to A. At time 7, T2 commits and GC=1 and TS(A)=1. At time 8, T1.2 starts (RV(T1.2)=1). At time 10, T1.2 attempts to write to A. By calling ValidateReaders in Figure 4.2, T1.2 validates itself and its ancestors (i.e., T1). T1.2 is valid because RV(T1.2)≥TS(A). T1 is not doomed because A is not yet in its read-set (i.e., T1.1's read-set has not been merged yet). Therefore, T1.2 can successfully write to A. At time 11, T1.1 merges its read-set to its parent's. At time 17, T1 successfully commits because it has an ownership for A (transferred from T1.2). However, this violates serializability because T1 eventually commits even when the two reads by T1.1 and T1.2 observe different versions of A.

Solution: The cause of this problem is that the commit process of T1.1 does not appear atomic to T1's descendants that validate T1 by calling ValidateReaders. To address this problem, we propose the *commit-lock* scheme. With this scheme, when a nested transaction attempts to commit, it must acquire the commit-lock of its parent. In addition, when a descendant validates its ancestor by calling ValidateReaders, it must also acquire the commit-lock of the validated ancestor. This ensures that the

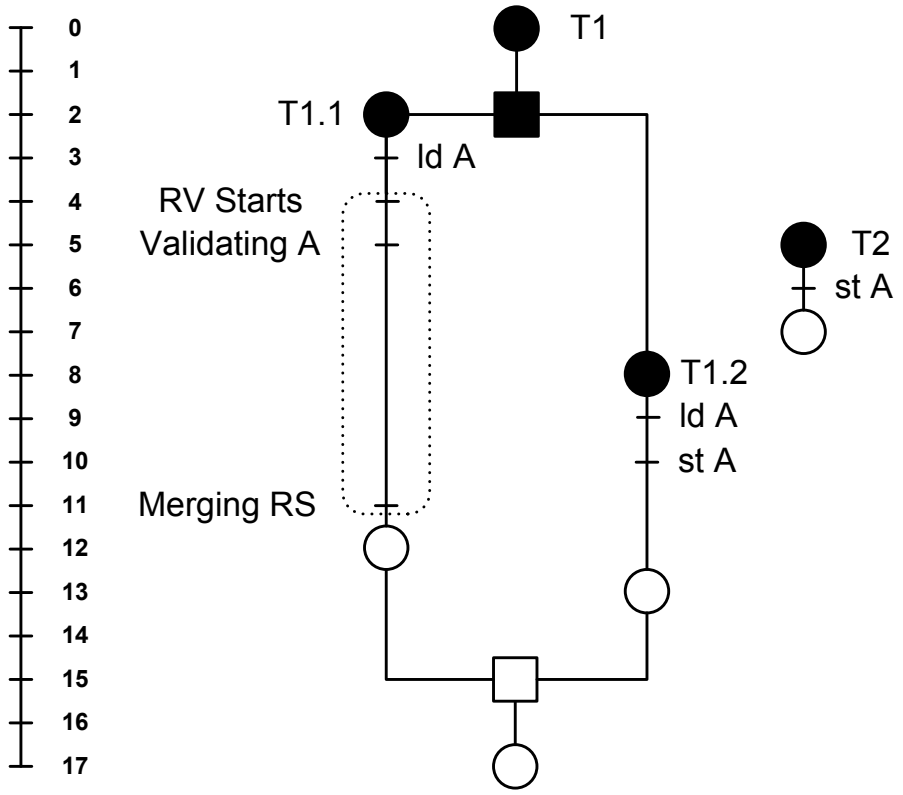


Figure 4.7: A potential serializability violation scenario due to the non-atomic commit.

commit process of a transaction's child appears atomic to a validating descendant of the transaction. In the previous example, with the commit-lock scheme, T1.1's commit either happens *before* or *after* the validation performed by T1.2. In the first case, T1 will be doomed because RV(T1)<TS(A) and eventually aborted. In the second case, T1.1 will be aborted because A is owned by T1.2 when T1.1 attempts to commit. Hence, no serializability violation occurs in both cases.

Performance impact: The commit-lock scheme essentially serializes the commits of child transactions. When a large number of child transactions simultaneously attempt to commit, performance can be significantly degraded due to the serialized commit.

Possible alternatives: We could also address this problem by introducing a *ValidationCounter* to each transaction. The ValidationCounter increments every time when a transaction is validated by its descendant. When a child transaction attempts to

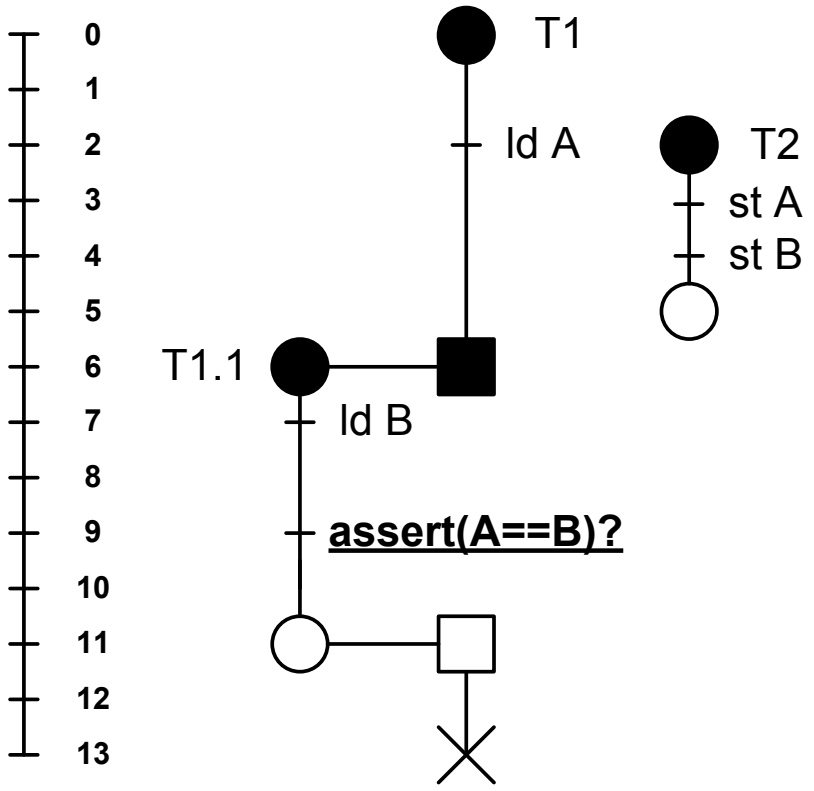


Figure 4.8: A problematic scenario due to a zombie transaction.

commit, it samples the value of the ValidationCounter of the parent. It then validates its read-set *without* acquiring the commit-lock of the parent. After the read-set validation, it acquires the commit-lock of the parent. It then compares the previously sampled value of the ValidationCounter with the current value. If the two values match, it can safely merge its read-set to its parent's because it is guaranteed that there has been no validation performed by any descendant of the parent. If the two values differ, it releases the commit-lock of the parent and conservatively repeats the whole process.

4.4.3 Zombie Transactions

Problem: Figure 4.8 illustrates a problematic scenario caused by a zombie transaction. Initially, GC=0 and TS(A)=TS(B)=0. At time 0, T1 starts (RV(T1)=0).

At time 2, T2 starts (RV(T2)=0). Then, T2 writes to A and B at times 3 and 4. At time 5, T2 commits and GC=1 and TS(A)=TS(B)=1. At time 6, T1.1 starts (RV(T1.1)=1). At time 7, T1.1 can successfully read B because B's owner is the root and RV(T1.1)≥TS(B). However, if a programmer assumes that A is always equal to B within transactions and inserts an assertion check, the program will be unexpectedly terminated by failing the assertion check. Note that if T1 could reach to its commit, it would eventually abort, thus no serializability violation. Other well-known anomalies such as infinite loops can also occur. The current NesTM implementation admits zombie transactions because we have not been able to find an efficient solution to avoid them in an unmanaged environment.

4.4.4 Correctness Status

At this point, we do not have a formal proof of the correctness (serializability) and liveness of the NesTM algorithm. Hence, the correctness and liveness of the NesTM algorithm still remain unchecked. However, we hope that our work will generate in-depth discussions on formally proving and verifying the correctness and liveness guarantees of timestamp-based STMs with support for concurrent nesting.

To establish some evidence of correctness, we have subjected the NesTM algorithm to exhaustive tests using our model checker (ChkTM) [11]. ChkTM verifies every possible execution of a small TM program running on the NesTM model. Refer to Chapter 6 for detailed information on ChkTM. We configured ChkTM to generate every possible program with four threads (i.e., [T1, T2, T1.1, T1.2]), each running only one transaction that performs at most two transactional memory operations (i.e., read or write), each accessing one of the two shared-memory words. ChkTM then explored every possible execution of every possible program. ChkTM, thus far, has not reported any serializability violation. With larger configurations (i.e., more threads or memory operations), however, ChkTM fails to verify NesTM due to the state space explosion.

Feature	Description
Processors	In-order, single-issue, x86 cores
L1 Cache	64-KB, 64-byte line, private 4-way associative, 1 cycle latency
Network	256-bit bus, split transactions pipelined, MESI protocol
L2 Cache	8-MB, 64-byte line, shared 8-way associative, 10 cycle latency
Main Memory	100 cycles latency up to 8 outstanding transfers

Table 4.2: Parameters for the simulated CMP system.

4.5 Evaluation

To study the performance of nested parallel transactions on NesTM, we use an execution-driven simulator for x86 multi-core systems. Table 4.2 summarizes architectural parameters. All operations, except for loads and stores, have a CPI of 1.0. However, all the details in the memory hierarchy timings are modeled, including the contention and queueing events. We used a microbenchmark (hashtable) that performs concurrent accesses to a hash table with 4K buckets. Among 4K operations, 12.5% are inserts (reads/writes) and 87.5% are look-ups (reads). The benchmark has 4 versions. flat uses only top-level transactions, each performing 8 operations. N1 pushes down the parallelism to NL=1, using the same code enclosed with one big outermost transaction[3]. N2 and N3 are implemented by repeatedly adding more outer-level transactions.

Figure 4.9 shows the execution time breakdowns of hashtable. The execution time of the microbenchmark is normalized to the execution time on an STM that flattens and serializes nested transactions (i.e., performs all 4K operations sequentially in a top-level transaction). Each bar has segments such as "busy" (useful instructions and cache misses), "RB" (read barriers), "WB" (write barriers), "aborted" (time spent on aborted transactions), "commit" (commit overhead), "CL" (time spent acquiring

[3]While flat and nested versions have different transactional semantics (i.e., whether to perform 4K operations atomically or not), we compare them to investigate performance issues.

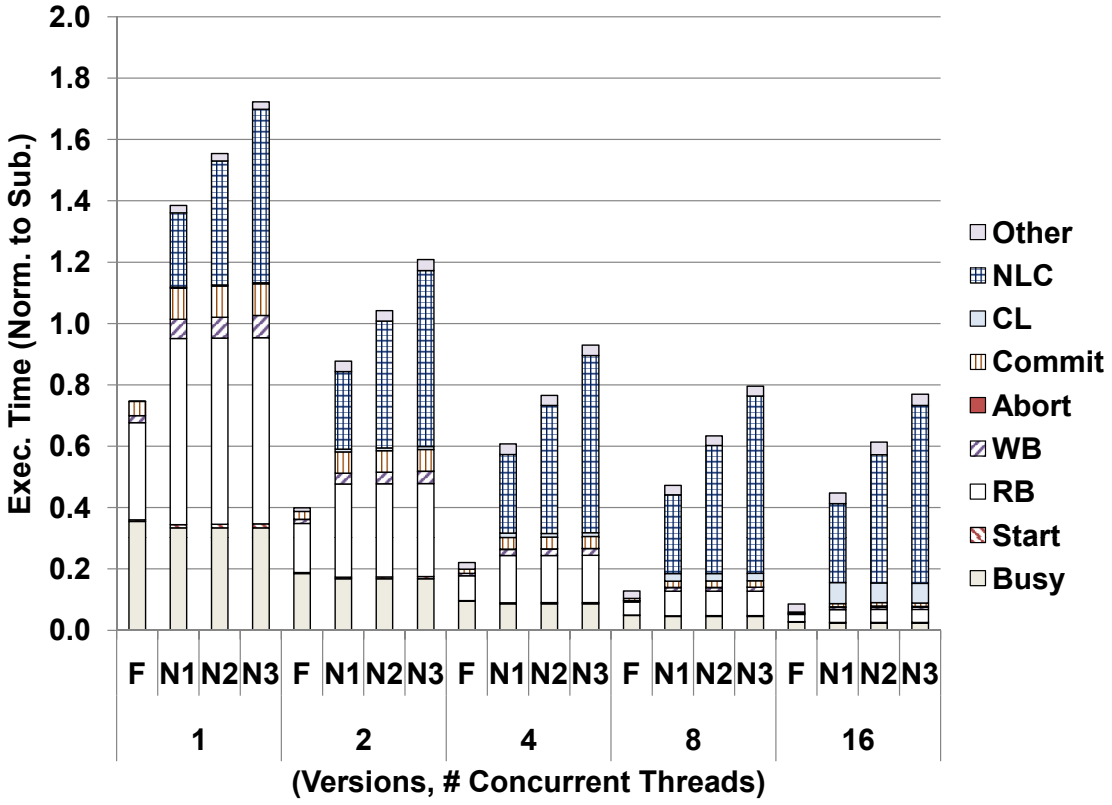


Figure 4.9: Execution time breakdowns of hashtable. Each bar shows the execution time normalized to that of the subsumed version.

the commit locks of parents), "NLC" (time spent committing non-leaf transactions), and "other" (work imbalance, etc.).

We observe that NesTM continues to scale up to 16 threads. For example, the N1 version of hashtable is faster than the subsumed version by 2.2× with 16 threads. Figure 4.9 also reveals the following major performance challenges in NesTM. First, the runtime overhead of the read and write barriers of nested transactions is more expensive than those of top-level transactions. This is mainly due to more cache misses when accessing the entries in read- and write-sets. Since previously-used entries in read- and write-sets of a transaction are merged to its parent, NesTM cannot exploit temporal locality on accessing the transactional metadata when it runs nested

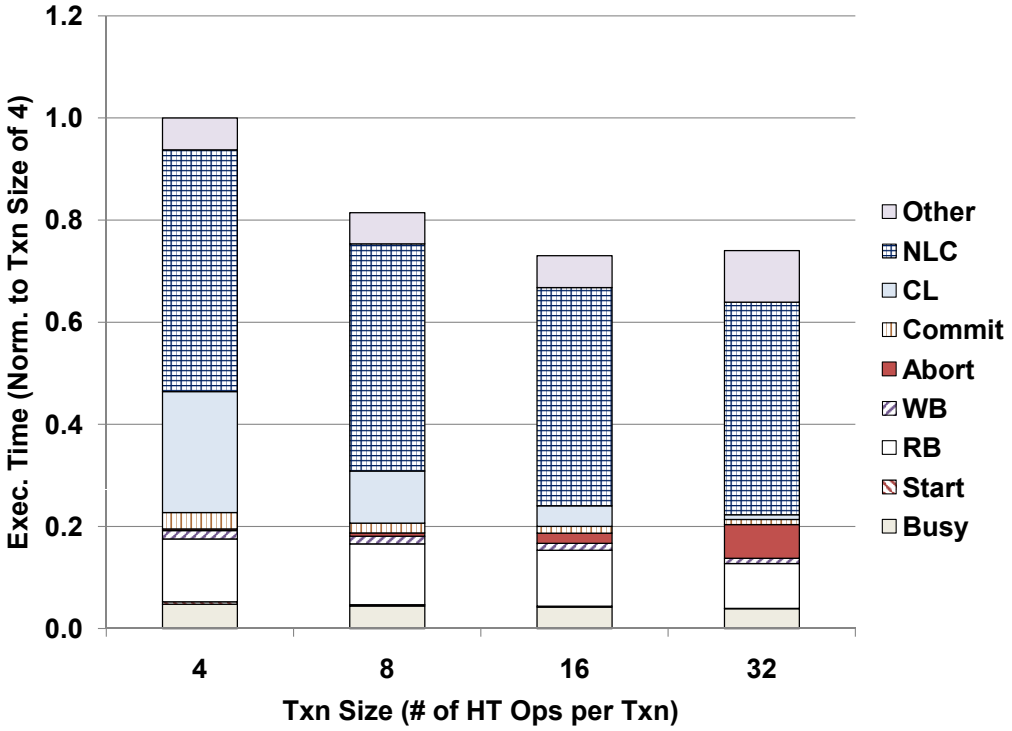


Figure 4.10: Execution time breakdowns of hashtable with various transaction sizes.

transactions. In contrast, when top-level transactions are used, there is significant locality in metadata accesses.

Second, commit time increases linearly with the nesting level mainly due to the repeated read-set validation across different nesting levels. Alternatively, a runtime may choose different policies (e.g., serialization, reader-lock scheme) depending on the nesting depth to achieve better performance. Finally, the contention on the commit-locks of parents can become a performance bottleneck when a large number of nested transactions simultaneously attempt to commit. Since conflicts are infrequent in hashtable even with 16 threads, many child transactions can simultaneously attempt to commit and trigger this lock contention. Chapter 5 discusses how these runtime overheads can be addressed using lightweight hardware support.

To understand the performance impact of transaction sizes, we measure the performance of hashtable by varying the transaction size from 4 to 32 operations per transaction. Figure 4.10 presents the normalized execution time with 16 threads. Each bar is normalized to the execution time with the transaction size of 4. With smaller transactions (e.g., 4), a significant portion of the time is spent on the commit-lock contention because a large number of (small) transactions simultaneously attempt to commit. With larger transactions (e.g., 32), while the performance overhead due to the commit-lock contention is mitigated, more time is wasted on aborted (large) transactions.

4.6 Related Work

Moss and Hosking described the reference model for closed and open nesting in transactional memory and discussed preliminary architectural sketches [72]. In addition, they proposed a simpler model called linear nesting in which nested transactions execute sequentially. There has been previous work on supporting linear nesting in HTM [66, 70] and STM [51, 74]. Our work differs since NesTM targets concurrent nesting.

Recently, there has been research on supporting nested parallelism in STM [8, 15, 80, 96]. Agrawal et al. proposed CWSTM, a theoretical STM algorithm that supports nested parallel transactions with the lowest upper bound of time complexity [8]. In [15], Barreto et al. proposed a practical implementation of the CWSTM algorithm. While achieving depth-independent time complexity of TM barriers, their work builds upon rather complex data structures such as concurrent stacks that could introduce additional runtime (especially to top-level transactions) and state overheads [15]. In contrast, NesTM extends a timestamp-based STM. Ramadan and Witchel proposed SSTM, a lazy STM-based design that supports nested parallel transactions [80]. Our

algorithm differs by extending an eager STM that has lower baseline overheads. Finally, Volos et al. proposed NePaLTM that supports nested parallelism inside transactions [96]. While efficiently supporting nested parallelism when no or low transactional synchronization is used, NePaLTM serially executes nested transactions using mutual exclusion locks. In contrast, NesTM targets concurrent nesting.

4.7 Conclusions

This chapter presented NesTM, an STM system that extends a high-performance, eager-versioning STM with closed-nested parallel transactions. NesTM is designed to keep the state overheads small. We also discussed the subtle correctness issues of concurrent nesting. Finally, we evaluated the performance of nested parallel transactions on NesTM. Chapter 5 discusses lightweight hardware support for improving the performance of nested parallel transactions.

Chapter 5

Hardware Support for Concurrent Nesting

5.1 Introduction

A few recent papers investigated the semantics of concurrent nesting and proposed prototype implementations in STM [8, 12, 15, 80, 96]. While compatible with existing chip multiprocessors, most STM implementations already suffer from the excessive runtime overheads of TM barriers even for single-level parallelism [26]. To make the problem even worse, supporting nested parallelism solely in software may introduce additional runtime overheads due to the use of complicated data structures [8, 15] or the use of an algorithm whose time complexity is proportional to the nesting depth [12]. For instance, as shown in the performance evaluation in this chapter, a single-threaded, transactional version of the red-black tree microbenchmark runs 6.2× slower with single-level transactions and 17.0× slower with nested transactions than a non-transactional, sequential version. Nested parallel transactions in STM will remain impractical unless these performance issues are successfully addressed.

A recent paper investigated how to support nested parallelism in HTM [93]. However, supporting nested parallelism solely in hardware may drastically increase hardware complexity, as it requires intrusive modifications to the hardware caches. For example, apart from the additional transactional metadata bits in tags, the design

proposed in [93] requires that the hardware caches are capable of maintaining multiple blocks with the same tag but different version IDs, and provide the version-combining logic that merges speculative data from multiple ways. Given the current trend in which major hardware companies are reluctant to introduce complicated hardware components to implement transactional functionality even for single-level parallelism, this hardware-only approach is unlikely to be adopted.

To address this problem, we propose *filter-accelerated, nested transactional memory* (FaNTM) that provides practical support for nested parallel transactions using hardware filters. FaNTM extends a baseline hybrid TM (SigTM) [26] to implement nesting-aware conflict detection and data versioning. FaNTM follows the transactional semantics of NesTM, but significantly improves the performance of nested transactions based on lightweight hardware support. Specifically, since hardware filters provide continuous, nesting-aware conflict detection, FaNTM effectively reduces the excessive runtime overheads of software nested transactions. In contrast to a full HTM approach, FaNTM simplifies hardware by decoupling nested transactions from the hardware caches. As a result, FaNTM makes nested parallel transactions practical in terms of both performance and implementation costs.

The specific contributions of this work are:

- We propose FaNTM, a hybrid TM system with support for concurrent nesting. FaNTM provides eager data versioning and conflict detection at the cache-line granularity across nested parallel transactions.

- We describe subtle correctness and liveness issues such as the *dirty-read* problem that do not exist in the non-nested baseline TM system. We also propose solutions to address the problems.

- We quantify the performance of FaNTM across multiple use scenarios. First, we show that the runtime overhead of FaNTM is small when applications use only single-level parallelism. Specifically, FaNTM is slower than the baseline hybrid TM by 2.3% on average when running STAMP applications. Second, we demonstrate that the incremental overhead of FaNTM for deeper nesting is reasonable. We also show that nested transactions on FaNTM run significantly

faster (e.g., 12.4×) than those on a nested STM. Finally, we demonstrate how FaNTM improves the performance of transactional applications using nested parallelism.

The rest of this chapter[1] is organized as follows. Section 5.2 provides background information. Section 5.3 presents FaNTM. Section 5.4 discusses subtle correctness and liveness issues. Section 5.5 quantifies the performance of FaNTM. Section 5.6 reviews related work, and Section 5.7 concludes this chapter.

5.2 Background

Semantics of Concurrent Nesting

FaNTM follows the same definitions and semantics of concurrent nesting as NesTM. Refer to Section 4.2 for the additional information on concurrent nesting.

The Baseline STM with Support for Concurrent Nesting

We use NesTM [12] as a proxy for a timestamp-based STM with support for concurrent nesting. We refer readers to Chapter 4 for the detailed information on NesTM.

Since all the nesting-aware transactional functionality is solely implemented in software, NesTM introduces substantial runtime overheads to nested transactions. One of the critical performance bottlenecks of NesTM is the repeated read-set validation, where the same memory objects must be repeatedly validated across different nesting levels [12]. Since this performance overhead increases linearly with the nesting depth, it limits the applications for which NesTM can improve performance [8, 12]. Furthermore, the NesTM barriers are more complicated, impacting performance even for top-level transactions. We quantify the performance differences between NesTM and FaNTM in Section 5.5.3.

[1]The work presented in this chapter was also published in [13].

The Baseline Hybrid TM

As our starting point, we use an eager-versioning hybrid TM system that follows the SigTM design [25, 26]. It uses hardware signatures to conservatively track read- and write-sets of transactions. Hardware signatures provide fast conflict detection at the cache-line granularity by snooping coherence messages. Data versioning is implemented in software using undo-logs. Refer to Section 2.4.3 for additional information on the eager-versioning SigTM design.

We chose to use eager versioning to avoid the runtime overheads of lazy versioning, which are higher for a nested TM. Since each update is buffered in the write buffer in a lazy TM, nested transactions must examine their parent's write buffer to correctly handle reads that follow a speculative write. These accesses are expensive, because they must be synchronized with the changes to the parent's write buffer when a sibling transaction commits. An eager-versioning TM does not require such look-ups because memory holds the speculative value.

5.3 Design of FaNTM

5.3.1 Overview

The baseline hybrid TM only supports single-level parallelism. Therefore, to support concurrent nesting, FaNTM hardware must be extended to provide nesting-aware conflict detection and retain multiple transactional contexts per processor core. FaNTM software must be extended to implement nesting-aware data versioning and handle liveness issues of concurrent nesting. This section describes our FaNTM design that implements high-performance nested parallel transactions in a manner that keeps hardware and software complexity low. We also provide a qualitative performance analysis on FaNTM.

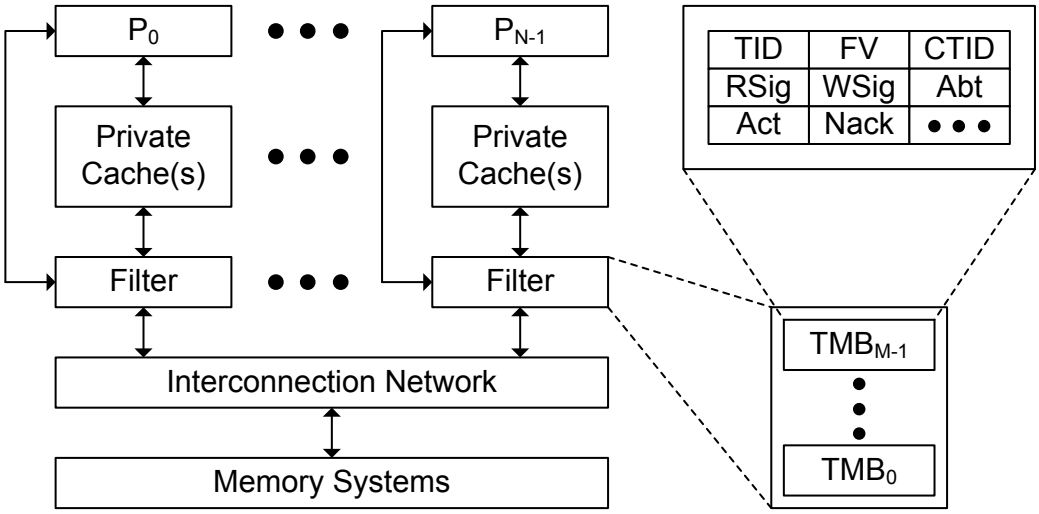


Figure 5.1: The overall architecture of FaNTM.

5.3.2 FaNTM Hardware

Figure 5.1 shows the overall architecture of FaNTM. Each processor core has a hardware filter. Filters are connected to the interconnection network to snoop coherence messages such as requests for shared and exclusive accesses and negative acknowledgements (nacks) to these requests. We assume that the underlying coherence protocol provides the nack mechanism. Filters may handle or propagate incoming messages to their associated cache depending on the characteristics of the messages. We also assume that each coherence message includes additional fields such as TID to encode the transactional information on the transaction that generated the message.

Each filter consists of a fixed number of *transactional metadata blocks* (TMBs) that summarize the state information of the transactions mapped on the corresponding processor. The number of TMBs in the filter limits the number of transactions that can be mapped on each processor without the need for virtualization. When the nesting depth overflows, we currently rely on software solutions such as switching back to a nested STM [12,15,80] or subsuming (i.e., serializing and flattening) nested transactions to avoid increasing hardware complexity. It is interesting future work to investigate hardware techniques for nesting depth virtualization.

State	Description
TID	T's TID
FV	A bit vector that encodes family(T). If a bit is set, the corresponding transaction belongs to family(T)
CTID	The TID of the transaction that conflicted with T
RSig	Read signature
WSig	Write signature
Abt	If set, T has a pending abort.
Act	If set, this TMB is the active TMB.
Nackable	If set, the nackable bit in outgoing memory requests is set.

Table 5.1: State information stored in each TMB. T denotes the transaction that is mapped on the TMB.

Message	Fields	Description
RSigMerge	destTid, RSig	Merge the read signature in the destination TMB and the RSig in the packet.
WSigMerge	destTid, WSig	Merge the write signature in the destination TMB and the WSig in the packet.
remoteFvSet	destTid, FV	Set each bit in FV in the destination TMB if the corresponding bit is set in the FV in the packet.
remoteFvReset	destTid, FV	Reset each bit in FV in the destination TMB if the corresponding bit is set in the FV in the packet.

Table 5.2: Filter-request messages used in FaNTM.

Table 5.1 summarizes the state information stored in each TMB. We discuss how each field is used as we describe FaNTM operations later.

Table 5.2 summarizes filter-request messages. The R/WSigMerge messages are used when committing nested transactions remotely merge their read/write signatures to their parent. The remoteFvSet/Reset messages are used when nested transactions remotely update their ancestor's FV. Note that every filter-request message is intercepted by filters, thus no need to modify the caches to handle the filter-request messages.

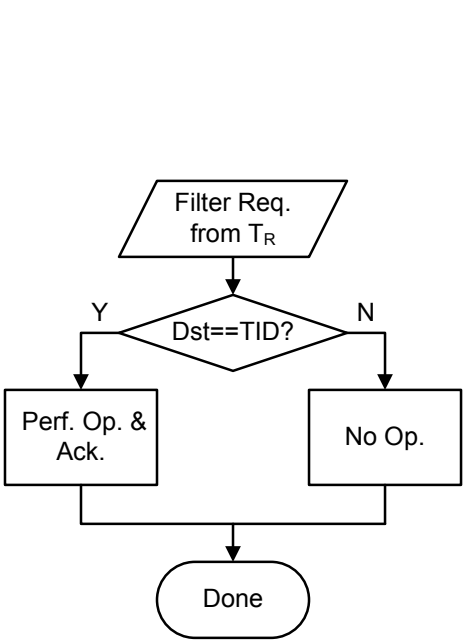
FaNTM Instruction	Description
`R/WSigReset`	Reset all the state in the read/write signature of the active TMB.
`R/WSigInsert` r1	Insert the address in register r1 in the read-/write signature of the active TMB.
`R/WSigEn/Disable`	Enable/Disable the read/write signature of the active TMB to look up coherence messages.
`R/WSigEnableNack` r1	Configure the read/write signature of the active TMB to nack conflicting requests (their types in register r1).
`fetchExclusive` r1	Send an exclusive load request for the address in register r1 over the network.
`mergeR/WSig` r1	Send a R/WSigMerge request with a destination TID in register r1 over the network.
`get/setTid` r1	Get/Set the TID value of the active TMB in register r1.
`getConfTid` r1	Get the CTID value of the active TMB in register r1.
`set/resetNackable`	Set/Reset the nackable bit of the active TMB.
`localFvSet/Reset` r1	Set/Reset the FV entry associated to a TID in register r1.
`remoteFvSet/Reset` r1	Send a remoteFvSet/Reset request with a destination TID in register r1 over the network.
`increaseNL`	Increase nesting level (NL) by allocating the TMB at one NL above and setting it as the active TMB.
`decreaseNL`	Decrease NL by releasing the current active TMB and setting the TMB at one NL below as the active TMB.

Table 5.3: User-level instructions in FaNTM.

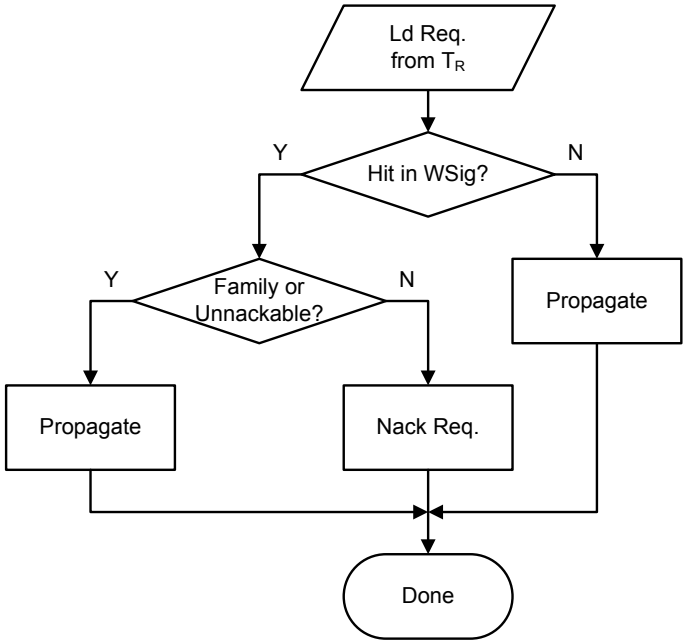
Table 5.3 summarizes the user-level instructions used to manipulate filters. The TID-related instructions are used to manipulate the TID of the active TMB. Outgoing memory requests are associated with this TID. The FV-related instructions are used to update the transactional hierarchy information. The `in/decreaseNL` instructions are used to switch TMBs when a nested transaction executes on the same core where its parent was running. We provide the details on the other instructions as we describe the FaNTM algorithm in Section 5.3.3.

Figure 5.2 illustrates the common-case TMB operations when receiving coherence messages. On receiving a filter-request message with a matching destination TID (Figure 5.2(a)), the TMB performs the requested operation. On receiving a shared-load request (Figure 5.2(b)), the TMB nacks the request if the requested address is contained in its write signature and the requesting transaction does not belong to its family. Note that the TMB does not nack the request if the nackable bit in the message is reset, which will be further discussed in Section 5.4.3. Otherwise, the TMB attempts to propagate the request to the associated cache. On receiving an exclusive-load request (Figure 5.2(c)), the TMB nacks it if the request satisfies the aforementioned nacking conditions. If the requested address is contained in the read signature (but not in the write signature), the following two cases are considered. First, if the request is from its family, the TMB attempts to propagate the request to the associated cache without aborting the transaction. Otherwise, the TMB sets its abt bit to eventually abort the transaction, disables its read signature to prevent repeated aborts, and resets its nackable bit to avoid deadlock (Section 5.4.3).

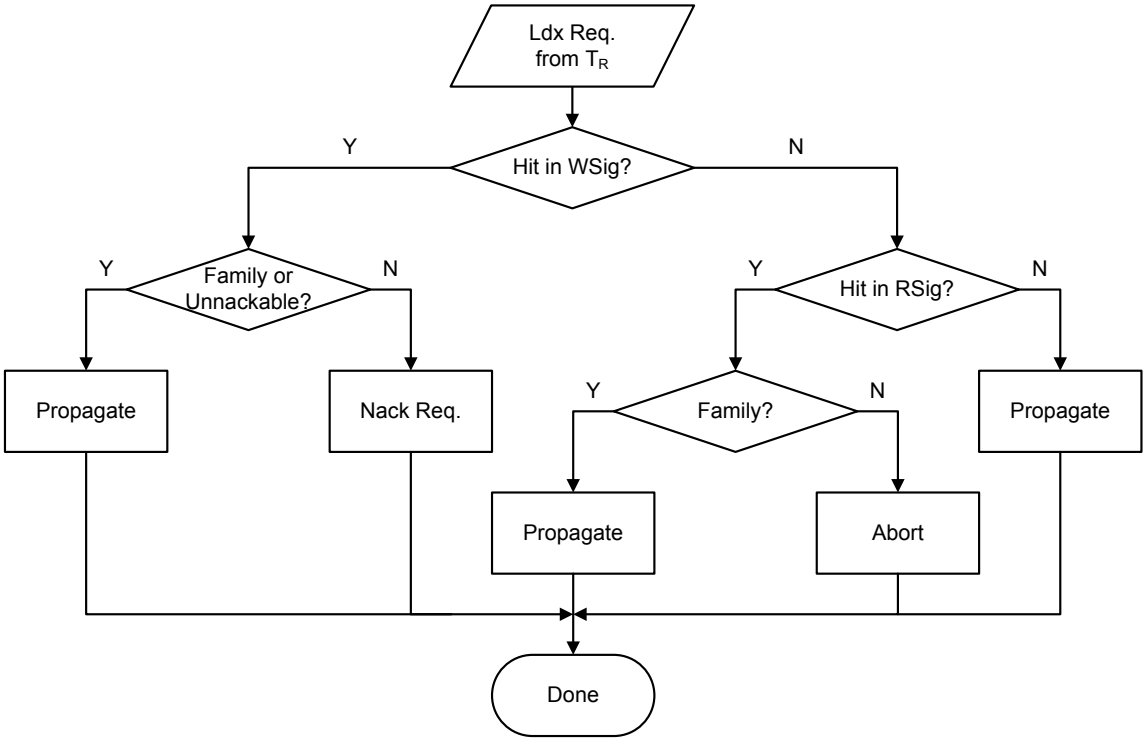
Figure 5.3 illustrates an example execution of a simple FaNTM program. T1 and T2 are top-level transactions running on P0 and P1. T3 is T1's child transaction running on P2. From step 0 to 2, T1 and T2 have performed transactional accesses to x and y. At step 3, P2 sends an exclusive load request prior to updating y. Since the request is from T1's family, Filter 0 simply propagates the request to the associated cache without aborting T1. On the other hand, since the request is not from T2's family (i.e., R/W conflict), Filter 1 interrupts P1 to abort T2. Since the exclusive load request by P2 is successful (not nacked), P2 acquires exclusive ownership for the cache line holding y and proceeds with the execution of T3.



(a) Filter request



(b) Shared-load request



(c) Exclusive-load request

Figure 5.2: Flowcharts of common-case TMB operations.

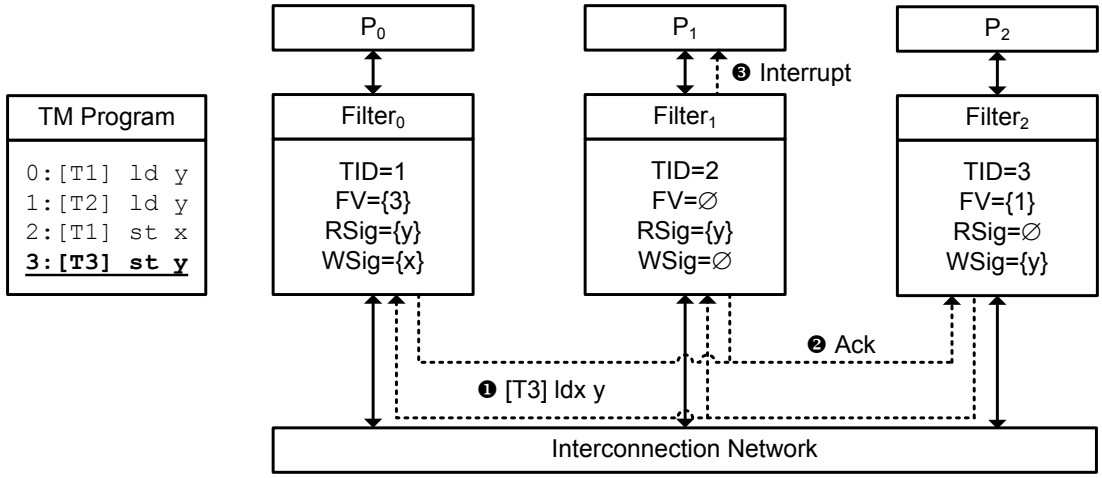


Figure 5.3: An example execution of a simple FaNTM program.

```
struct transaction {
 int Tid;
 Log UndoLog;
 struct transaction* Parent;
 lock commitLock;
 bool Doomed;
 int Aborts;
 bool Active;
 ...
}
```

Figure 5.4: The transaction descriptor.

5.3.3 FaNTM Software

Figure 5.4 shows the transaction descriptor, a software data structure that summarizes the transactional metadata. Each transaction maintains an undo-log implemented using a doubly-linked list to provide eager data versioning in software. It has a pointer to its parent transaction to access its parent's metadata as necessary. Each transaction maintains a commit-lock to synchronize concurrent accesses to its undo-log by its children. There are additional data fields such as Doomed, Aborts, and Active, which will be discussed later.

```
1: procedure DoomConflictingAnces(Self, T)
2:   p ← Self.Parent
3:   while p ≠ NIL and IsAnces(p,T) = false  do
4:     p.Doomed ← true
5:     p ← p.Parent
6:   return

7: procedure TxStart(Self)
8:   checkpoint()
9:   if isDoomedAnces(Self) then
10:    return failure
11:  Self.Doomed ← false
12:  Self.Active ← true
13:  enableRSigLookup()
14:  enableWSigLookupAndNack()
15:  setNackable()
16:  return success

17: procedure TxLoad(Self, addr)
18:  RSigInsert(addr)
19:  val ← Memory[addr]
20:  return val
21: procedure TxStore(Self, addr, data)
22:  WSigInsert(addr)
23:  fetchExclusive(addr)
24:  Self.UndoLog.insert(addr, Memory[addr])
25:  Memory[addr] ← data
26:  return

27: procedure TxAbortHandler(Self)
28:  if Self.Aborts%period = period − 1 then
29:    confTx ← getConfTx()
30:    DoomConflictingAnces(Self, confTx)
31:  if Self.Active = false then
32:    Self.Doomed ← true
33:  else
34:    initiateAbort(Self)
```

Figure 5.5: Pseudocode for the FaNTM algorithm.

```
 1: procedure TxCommit(Self)
 2:   resetNackable()
 3:   if isDoomedOrDoomedAnces(Self) then
 4:     TxAbort(Self)
 5:   if isTopLevel(Self) = false then
 6:     mergeRSigToParent(Self)
 7:     disableRSigLookup()
 8:     mergeWSigToParent(Self)
 9:     acquireCommitLock(Self.Parent)
10:     mergeUndoLogToParent(Self)
11:     releaseCommitLock(Self.Parent)
12:   else
13:     disableRSigLookup()
14:   RSigReset()
15:   WSigReset()
16:   disableWSigLookup()
17:   Self.UndoLog.reset()
18:   Self.Aborts ← 0
19:   return

20: procedure TxAbort(Self)
21:   resetNackable()
22:   disableRSigLookup()
23:   RSigReset()
24:   for all addr in Self.UndoLog do
25:     Memory[addr] ← Self.UndoLog.lookup(addr)
26:   WSigReset()
27:   disableWSigLookup()
28:   Self.UndoLog.reset()
29:   Self.Aborts ← Self.Aborts + 1
30:   Self.Doomed ← false
31:   doContentionManagement()
32:   restoreCheckpoint()
```

Figure 5.6: Pseudocode for the FaNTM algorithm.

Figures 5.5 and 5.6 present the pseudocode for the FaNTM algorithm. We summarize its key functions below.

TxStart: After taking a checkpoint, this barrier checks whether there is any doomed ancestor. If there is any doomed ancestor, it returns "failure" to initiate recursive aborts to provide forward progress. Otherwise, it starts the transaction by initializing its metadata. Note that the nackable bit in the TMB is set to ensure that any conflicting memory access by the transaction is correctly handled.

TxLoad[2]: This barrier inserts the address in the read signature and attempts to read the corresponding memory object. If this load request is successful (i.e., not nacked), the read barrier returns the memory value. Otherwise, the processor is interrupted and the program control is transferred to the software abort handler (TxAbortHandler) to initiate an abort.

TxStore: This barrier inserts the address in the write signature. It then sends an exclusive load request for the address over the interconnection network using the fetchExclusive instruction (Table 5.3). If this request fails, the filter interrupts the processor to abort the transaction. Otherwise, the transaction inserts the current memory value into its undo-log and updates the memory object in-place.

TxCommit: This barrier first resets the nackable bit in the active TMB to handle the deadlock issues discussed in Section 5.4.3. If there is any doomed ancestor, a transaction aborts. Otherwise, a top-level transaction finishes its commit by resetting its metadata. A nested transaction merges its read signature into its parent by sending a RSigMerge message. The nested transaction should detect any potential conflict until it receives the ack for the RSigMerge message from its parent. After receiving the ack from its parent, the nested transaction can disable its read signature because its parent will detect any subsequent conflict on behalf of the transaction. The nested transaction then merges its write signature by sending a WSigMerge message and its undo-log to its parent. When merging its undo-log, the nested transaction must

[2]Similar to TxStart, TxLoad (and TxStore) could also check doomed ancestors. There are three possible schemes to check doomed ancestors such as always, periodic, and never. The always and periodic schemes introduce a runtime overhead that increases linearly with the nesting depth. The never and periodic schemes admit an execution scenario in which a nested transaction keeps running (temporarily), even when its ancestor has been already doomed. In our performance evaluation (Section 5.5), we performed experiments with a FaNTM design that implements the never scheme.

TM Barrier	TL2	SigTM	NesTM	FaNTM
Read	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Write	$O(1)$	$O(1)$	$O(d \cdot R)$	$O(1)$
Commit	$O(R+W)$	$O(1)$	$O(d+R+W)$	$O(d)$

Table 5.4: A symbolic comparison of the time complexity of the performance-critical TM barriers in (eager) TL2, (eager) SigTM, NesTM, and FaNTM. R, W, and d indicate the read-set size, write-set size, and nesting depth, respectively.

acquire its parent's commit-lock to avoid data races. To reduce the execution time in the critical section, undo-log entries are merged by linking the pointers (instead of copying the entries). Finally, the transaction finishes its commit by resetting the transactional metadata.

TxAbort: This barrier restores the speculatively written memory values. It then resets the transactional metadata including the write signature. After performing contention management (e.g., exponential backoff), the transaction restarts by restoring the checkpoint.

TxAbortHandler: This software interrupt handler is pre-registered. It is invoked when a processor is interrupted due to a conflict. To handle the liveness issue discussed in Section 5.4.2, the transaction periodically dooms its ancestors by setting their Doomed variable. If the transaction is currently inactive (i.e., has live child transactions), it sets its Doomed variable and defers the actual abort until it becomes active again. Otherwise, the transaction initiates an abort.

5.3.4 Qualitative Performance Analysis

Table 5.4 presents a symbolic comparison of the time complexity of the TM barriers in TL2, SigTM, NesTM, and FaNTM. Note that all the TMs discussed here perform eager data versioning. The time complexity of the read barrier in all TMs is $O(1)$[3]. The time complexity of the write barrier in NesTM is high ($O(d \cdot R)$) because a transaction should validate its ancestors before it updates a memory object. In contrast, all the other TMs have $O(1)$ complexity. As for the commit barrier, SigTM has the

[3]We assume that transactions do not check doomed ancestors in the read and write barriers of NesTM and FaNTM.

fastest one because committing transactions simply reset their metadata. FaNTM has $O(d)$ complexity because it checks doomed ancestors. TL2 has $O(R+W)$ complexity because a committing transaction validates its read-set and releases the acquired locks in its write-set. NesTM has slightly higher complexity ($O(d+R+W)$) because it checks doomed ancestors.

Apart from the differences in the time complexity, there are additional FaNTM performance issues to note. First, nested transactions cannot exploit temporal locality when accessing their undo-log entries. When a nested transaction commits, it merges its undo-log entries into its parent. Therefore, temporal locality is lost for these entries when a new nested transaction starts on the same core. Second, when a large number of child transactions commit at the same time, the contention on the commit-lock of their parent may degrade performance. Finally, the extra code in TM barriers (e.g., checking doomed ancestors, sending R/WSigMerge messages over the interconnection network when nested transactions commit) may introduce additional runtime overheads. We quantify their performance impact in Section 5.5.

5.4 Complications of Concurrent Nesting

In this section, we discuss subtle correctness and liveness issues encountered while developing FaNTM.

5.4.1 Dirty Read

Problem: The key assumption for guaranteeing the correctness of FaNTM is that any transactional memory access that conflicts with other transactions causes a cache miss. Filters then snoop the conflicting request and correctly resolve it. In the presence of nested parallel transactions, however, this assumption may not hold if threads are carelessly scheduled. Figure 5.7 illustrates the *dirty read* problem that may occur when this assumption does not hold. At step 1, T1 on P0 attempts to write to `x`. P0's cache acquires exclusive ownership for the line holding `x` and the corresponding lines in the other caches are invalidated. At step 3, T1.1 on P1

Assume that initially `x==0`

```
   // Running on P0    // Running on P1
0: Start(T1)          ...
1: st x,1             ...
2: ...                Start(T1.1)
3: ...                ld x // cache miss
4: ...                Commit(T1.1)
5: ...                // Thr1.1 fin./Thr2 starts
6: ...                Start(T2)
7: ...                ld x // cache hit!
8: ...                Commit(T2)
9: // T1 aborts       ...
```

Can T2 observe `x==1`?

Figure 5.7: A dirty-read problem due to an unexpected cache hit.

attempts to read `x`. This access causes a cache miss due to the prior invalidation. Since T1.1 belongs to T1's family, T1 acks the request. Hence, the cache lines holding `x` in P0's and P1's caches are now in the shared state. After T1.1 commits, another top-level thread is scheduled on P1 and executes T2. At step 7, T2 attempts to read `x`. While this access conflicts with T1, it cannot be nacked by T1 due to an unexpected cache hit. At step 8, T2 successfully commits even after it read the value speculatively written by T1, which is incorrect.

Solution: The root cause of this problem is that unexpected cache hits can occur when a potentially-conflicting transaction (T2 in Figure 5.7) runs on a processor where a nested transaction (T1.1) ran and its top-level ancestor (T1) has not been quiesced yet. To address this problem without requiring complex hardware, we rely on a software thread-scheduler approach. If the number of available processors in the system is no less than the number of threads in the application, the thread scheduler pins each thread on its dedicated processor[4]. If the number of processors is not large enough, the thread scheduler attempts to schedule a thread on a processor where a

[4]In our performance evaluation (Section 5.5), we assumed that the number of available processors is no less than the number of threads.

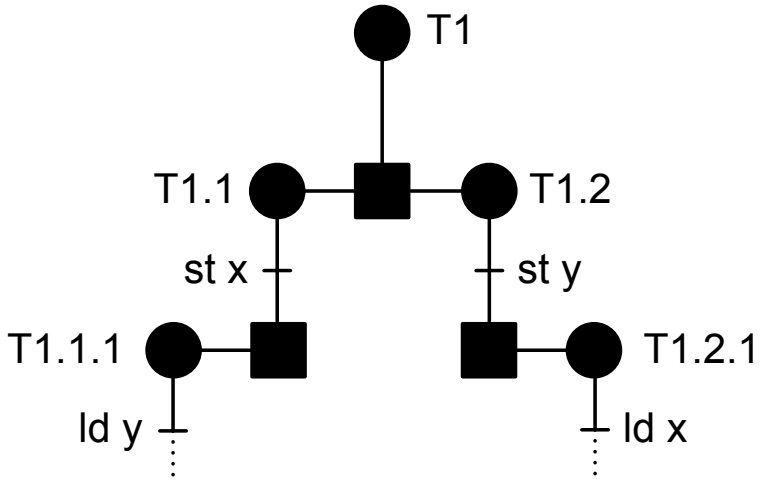


Figure 5.8: A livelock scenario avoided by eventual rollback of the outer transaction.

```
// T1 running on P0
atomic {
 ...
 fork(...);


}
```

```
// T1.2 running on P1
work(arg) {
 // Any memory access can be
 // potentially nacked until
 // TID is set.
 setTid(tid);
}
```

Figure 5.9: A self-deadlock scenario caused by carelessly enforcing strong isolation.

transaction whose family has been quiesced ran. If there is no such processor, the thread scheduler maps the thread to any processor. When the thread is about to start a transaction, it may defer its execution until the family of the previously-executed transaction is quiesced or invalidate the private cache to prevent unexpected cache hits. We leave an exhaustive exploration of the software thread-scheduler approach as future work.

5.4.2 Livelock

Problem: When a nested transaction detects a conflict, it only aborts and restarts (instead of aborting its ancestors) to avoid any unnecessary performance penalty.

```
   // Running on P0          // Running on P1
0: TxStart(T1)              TxStart(T2)
1: TxStore(x,1)             TxStore(y,2)
2: TxLoad(y)                TxLoad(x)
3: // TxAbort               // TxAbort
4: access UndoLog(T1,x)     access UndoLog(T2,y)
5: // No progress           // No progress
```

Figure 5.10: A deadlock scenario caused by carelessly enforcing strong isolation.

However, this can potentially lead to livelock scenarios. Figure 5.8 illustrates an example. If only nested transactions (i.e., T1.1.1 and T1.2.1) abort and restart, none of them can make forward progress because the memory objects are still (crosswise) locked by their ancestors (i.e., T1.1 and T1.2).

Solution: To address this problem, a nested transaction periodically dooms its ancestors when executing TxAbortHandler (line 30 in Figure 5.5). Specifically, when a TMB detects a conflict, it records the TID of the conflicting transaction in its CTID. Using the CTID, the transaction periodically dooms every ancestor that is not the ancestor of the conflicting transaction. For example, when T1.1.1 in Figure 5.8 is aborted by T1.2, it dooms its parent (T1.1) because T1.1 is not T1.2's ancestor. However, T1 is not doomed because it is T1.2's ancestor. If T1 is carelessly doomed, it causes a *self livelock* where a transaction cannot make forward progress as it is aborted by its descendants, even without any conflicting transaction.

5.4.3 Deadlock

Problem: In line with its baseline TM system, FaNTM provides strong isolation where a transaction is isolated both from other transactions and non-transactional memory accesses [61]. However, carelessly enforcing strong isolation may cause various deadlock issues due to the inexact nature of the hardware signatures. Figures 5.9 and 5.10 illustrate potential deadlock scenarios. In Figure 5.9, any memory access performed by T1.2 on P1 (before TID is set) can be potentially nacked by T1 on P0, if it generates a false positive in T1's write signature. T1.2 is nacked by T1 which is

re-activated only after T1.2 finishes, creating a cycle in the dependency graph. Thus, a self deadlock occurs.

Note that deadlock can occur even in the baseline hybrid TM system. Figure 5.10 illustrates another deadlock scenario. T1 on P0 and T2 on P1 abort each other, after accessing memory objects (`x` and `y`) in a crosswise manner. When T1 and T2 attempt to restore the memory objects, they can potentially deadlock, if their memory accesses are nacked by each other due to the false positives in the write signatures.

Solution: To address this problem, we enforce the following rules. First, by default, every non-transactional memory request is associated with the TID 0 (i.e., the root transaction). Since, by definition, the root transaction belongs to the family of every transaction, filters do not respond to the request with the TID 0. If strong isolation is desired, it should be explicitly enabled by associating non-transactional memory accesses with a non-zero TID using the `setTid` instruction in Table 5.3. Second, every memory request by a committing or aborting transaction resets its nackable bit. Filters do not nack this memory request even if it hits in their write signatures.

5.5 Evaluation

5.5.1 Methodology

We used an execution-driven simulator for x86 multi-core systems. Table 5.5 summarizes the main architectural parameters. The processor model assumes that all instructions have a CPI of 1.0 except for the instructions that access memory or generate messages over the interconnection network. However, all the timing details in the memory hierarchy are modeled, including the contention and queueing events.

Through our performance evaluation, we aim to answer the following three questions: **Q1**: What is the performance overhead of FaNTM when transactional applications only use top-level parallelism? **Q2**: What is the performance overhead when the available parallelism is exploited in deeper nesting levels? **Q3**: How can we exploit nested parallelism to improve transactional application performance?

Feature	Description
Processors	In-order, single-issue, x86 cores
L1 Cache	64-KB, 64-byte line, private 4-way associative, 1 cycle latency
Network	256-bit bus, split transactions Pipelined, MESI protocol
L2 Cache	8-MB, 64-byte line, shared 8-way associative, 10 cycle latency
Main Memory	100 cycle latency up to 8 outstanding transfers
Filters	4 TMBs per filter 2048 bits per R/W signature register The hash functions reported in [26]

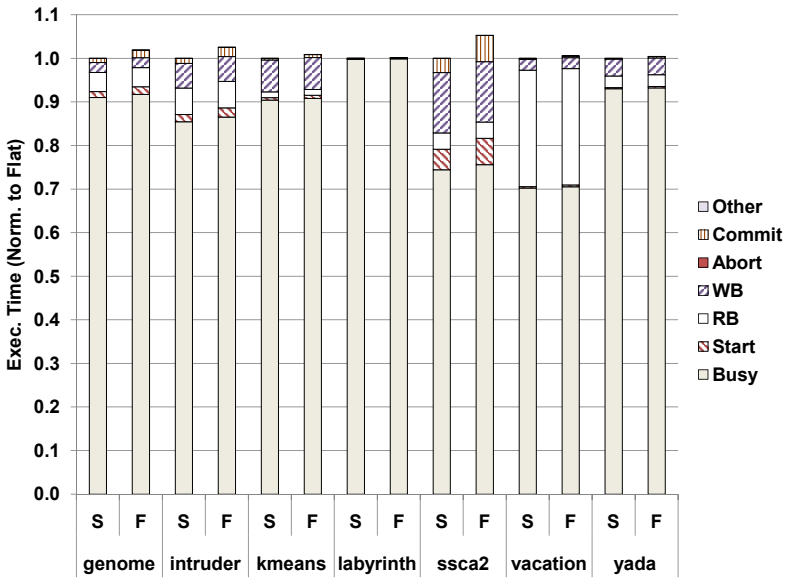
Table 5.5: Parameters for the simulated CMP system.

For Q1, we used seven STAMP benchmarks [25]. For Q2, we used two microbenchmarks based on concurrent hash table (hashtable) and red-black tree (rbtree). For Q3, we used a benchmark (np-rbtree) that uses multiple red-black trees and the labyrinth application from the STAMP benchmark suite. We provide more details on the benchmarks later in this section.

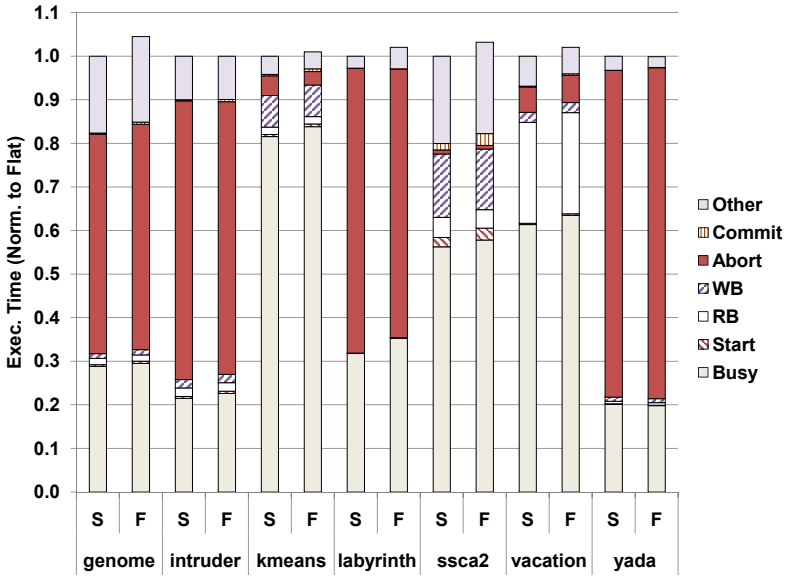
5.5.2 Q1: Overhead for Top-Level Parallelism

Table 5.6 demonstrates the performance differences between SigTM and FaNTM using the STAMP benchmarks that only use top-level transactions. It summarizes the normalized performance difference (NPD) defined as $NPD(\%) = \frac{T_{FaNTM} - T_{SigTM}}{T_{SigTM}} \times 100$ (the larger NPD, the slower FaNTM).

As shown in Table 5.6, the average NPD is 2.3% across all the benchmarks and thread counts, which is small. While the FaNTM barriers such as TxStart and TxCommit include extra code, their performance impact is insignificant because they are infrequently executed (compared to TxLoad and TxStore). To understand the exact runtime overheads, we show execution time breakdowns in Figures 5.11(a) and 5.11(b). The execution time of each application is normalized to the one on SigTM with 1 thread (Figure 5.11(a)) and 16 threads (Figure 5.11(b)). Execution time is



(a) 1 thread



(b) 16 threads

Figure 5.11: Execution time breakdowns of STAMP applications. S and F indicate SigTM and FaNTM.

# T	G	I	K	L	S	V	Y	Avg.
1	1.9	2.5	0.9	0.1	5.3	0.6	0.4	1.7
2	7.3	4.1	1.6	0.2	6.1	0.6	2.6	3.2
4	1.2	0.0	0.8	1.2	6.7	1.3	5.7	2.4
8	1.5	1.8	1.7	2.5	5.3	3.4	0.4	2.4
16	4.5	0.0	1.0	2.0	3.2	2.0	0.0	1.8
Avg.	3.3	1.7	1.2	1.2	5.3	1.6	1.8	2.3

Table 5.6: Normalized performance difference (%) of FaNTM relative to SigTM for STAMP applications. G, I, K, L, S, V, and Y indicate genome, intruder, kmeans, labyrinth, ssca2, vacation, and yada, respectively. The rightmost column and the bottommost row present average values.

broken into "busy" (useful instructions and cache misses), "start" (TxStart overhead), "RB" (read barriers), "WB" (write barriers), "abort" (time spent on aborted transactions), "commit" (TxCommit overhead), and "other" (work imbalance, etc.).

With 1 thread (Figure 5.11(a)), NPD is relatively high (FaNTM is slower) when small transactions are used and account for a significant portion of the execution time (e.g., intruder and ssca2) [25]. This is because the runtime overhead due to the extra code in TxStart and TxCommit is not sufficiently amortized with small transactions. On the other hand, when larger transactions are used (e.g., labyrinth, vacation, and yada), the performance difference becomes small. We observe a similar performance trend with 16 threads (Figure 5.11(b)) except that several applications spend a significant portion of the execution time on aborted transactions.

5.5.3 Q2: Overhead of Deeper Nesting

We quantify the incremental runtime overhead when the available parallelism is used in deeper nesting levels (NLs). We used two microbenchmarks: hashtable and rbtree. They perform concurrent operations to a hash table (hashtable) with 4K buckets and a red-black tree (rbtree). hashtable performs 4K operations where 12.5% are inserts (reads/writes) and 87.5% are look-ups (reads). rbtree performs 4K operations where 6.25% are inserts (reads/writes) and 93.75% are look-ups (reads). Each transaction in hashtable and rbtree performs 8 and 4 operations. Each microbenchmark has 4

versions. The flat version uses only top-level transactions. N1 pushes down the available parallelism to NL=1, by enclosing the same code with a big outer transaction. We implemented N2 and N3 by adding more outer-level transactions in a repeated manner. Note that flat and nested versions have different transactional semantics (i.e., whether to perform all the operations atomically or not). We compare them to investigate the runtime overheads of nested transactions.

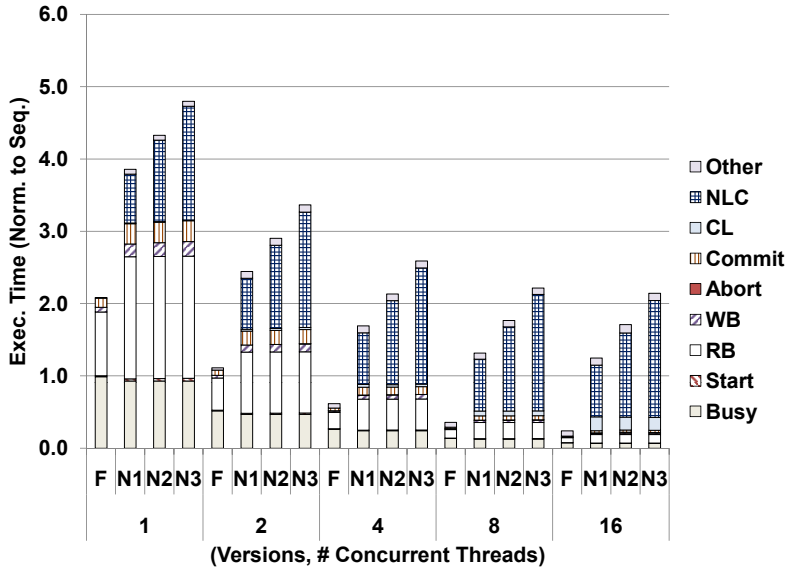
Figures 5.12 and 5.13 show the execution time breakdowns of hashtable and rbtree normalized to that with the non-transactional, sequential version. Figures 5.12(a) and 5.13(a) show the results with NesTM, while Figures 5.12(b) and 5.13(b) present the results with FaNTM. Apart from the aforementioned segments, each bar contains additional segments: "CL" (time spent acquiring the commit-locks of parents), and "NLC" (time spent committing non-leaf transactions).

We aim to answer the following three sub-questions: **Q2-1**: What is the incremental runtime overhead of nested parallel transactions over top-level transactions? **Q2-2**: How much faster is FaNTM than a nested STM (NesTM) when running nested parallel transactions? **Q2-3**: How much computational workload is required to amortize the overhead of deeper nesting?

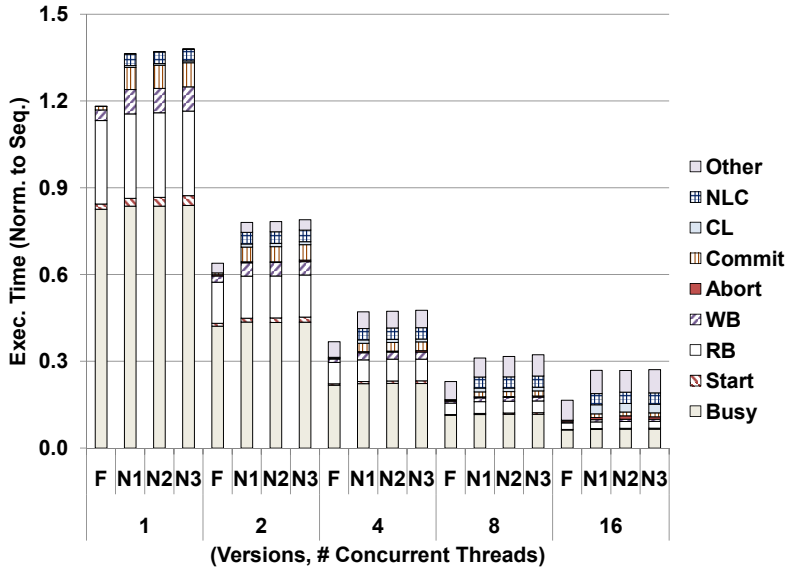
Q2-1: *What is the incremental performance overhead of nested parallel transactions over top-level transactions?*

Figures 5.12(b) and 5.13(b) show that FaNTM continues to scale up to 16 threads. With 16 threads, N1 versions are faster than the sequential version by 3.6× (hashtable) and 2.2× (rbtree). Scalability of rbtree with 16 threads is limited by conflicts among nested transactions. Figures 5.12(b) and 5.13(b) also reveal the following FaNTM performance issues. First, the runtime overhead of the write barrier becomes more expensive when running nested transactions. This is due to more cache misses occurred when accessing undo-log entries. Since previously-used undo-log entries of a nested transaction are merged to its parent, temporal locality is lost when accessing these entries. However, since writes are relatively infrequent (compared to reads) in both microbenchmarks, the performance impact is not significant.

Second, performance can be degraded due to the contention on the commit-lock of the parent when a large number of nested transactions simultaneously attempt to

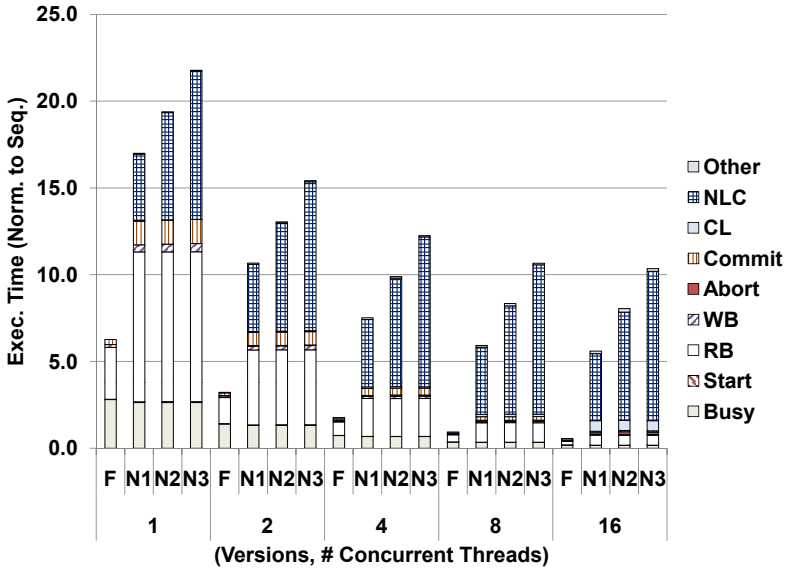


(a) NesTM (software)

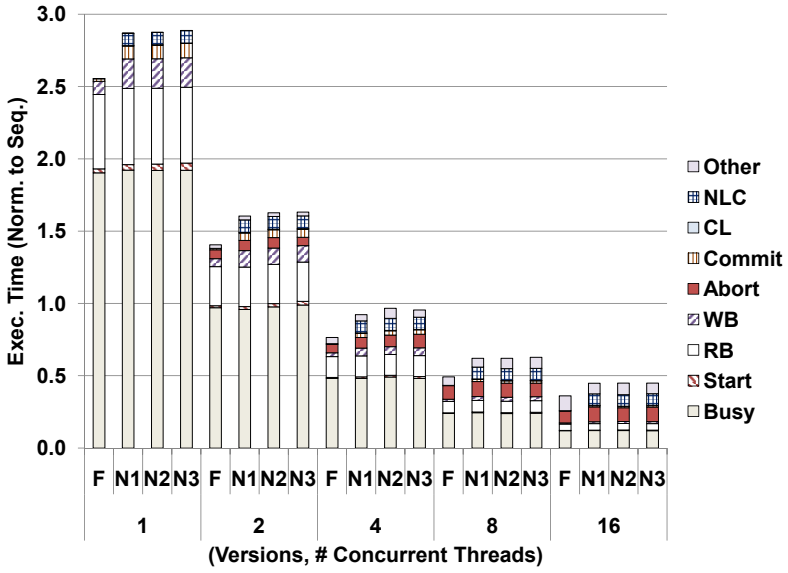


(b) FaNTM (hybrid)

Figure 5.12: Execution time breakdowns of hashtable at various nesting levels.



(a) NesTM (software)



(b) FaNTM (hybrid)

Figure 5.13: Execution time breakdowns of rbtree at various nesting levels.

commit. In hashtable, nested transactions rarely conflict with each other even with 16 threads (Figure 5.12(b)). Hence, many child transactions can simultaneously attempt to commit and cause this commit-lock contention. In contrast, conflicts among nested transactions are relatively frequent in rbtree with 16 threads (Figure 5.13(b)). Thus, the performance impact of this commit-lock contention is not significant. Finally, the extra code in TxStart and TxCommit may introduce additional runtime overheads. However, since they are amortized using reasonably large transactions in both microbenchmarks, their performance impact is not critical.

Q2-2: *How much faster is FaNTM than NesTM when running nested parallel transactions?*

Figures 5.12(a) and 5.12(b) (and also Figures 5.13(a) and 5.13(b)) demonstrate that FaNTM significantly outperforms NesTM, when running nested transactions. For example, N1 versions with 16 threads run 4.6× (hashtable) and 12.4× (rbtree) faster on FaNTM than NesTM. This performance improvement is achieved by addressing the two critical performance issues of NesTM.

First, FaNTM eliminates the linearly-increasing runtime overheads of NesTM such as the repeated read-set validation. Since NesTM repeatedly validates the same memory objects in the read-set across different nesting levels, it suffers from excessive runtime overheads that linearly increase with the nesting depth. In contrast, since hardware filters continuously provide nesting-aware conflict detection, FaNTM does not suffer from this performance pathology.

Second, the performance of the FaNTM read barrier is almost unaffected when running nested transactions, whereas the performance of the NesTM read barrier is significantly degraded. This is mainly due to more cache misses occurred when accessing read-set entries in NesTM. Since committing transactions merge their read-set entries to their parent, NesTM cannot exploit temporal locality when accessing these entries. In contrast, since the software read-sets are replaced with the hardware signatures, FaNTM does not suffer from this performance pathology.

Q2-3: *How much computational workload is required to amortize the overhead of deeper nesting?*

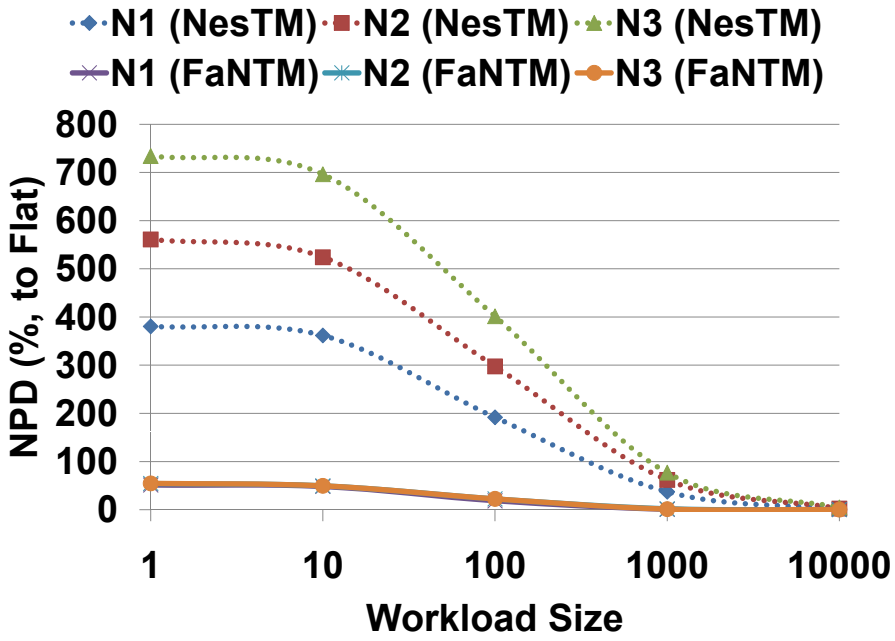


Figure 5.14: Performance sensitivity to the workload size.

We investigate how much computational workload is required to amortize the runtime overheads of nested transactions on FaNTM and NesTM. To this end, we compare the performance of the nested versions of hashtable with the flat version by varying the size of computational workload within transactions. Figure 5.14 presents the normalized performance difference (NPD) with 16 threads. With little work, NPD is very high on NesTM because the extra runtime overheads of nested transactions cannot be amortized. On the other hand, even with little work, the nested versions on FaNTM perform comparably with the flat version because FaNTM introduces reasonable runtime overheads to nested transactions. Furthermore, as the nesting depth increases, NesTM requires even larger workloads to amortize the linearly-increasing overheads. In contrast, the performance of nested transactions on FaNTM is almost unaffected by nesting depth, requiring no extra workload for deeper nesting.

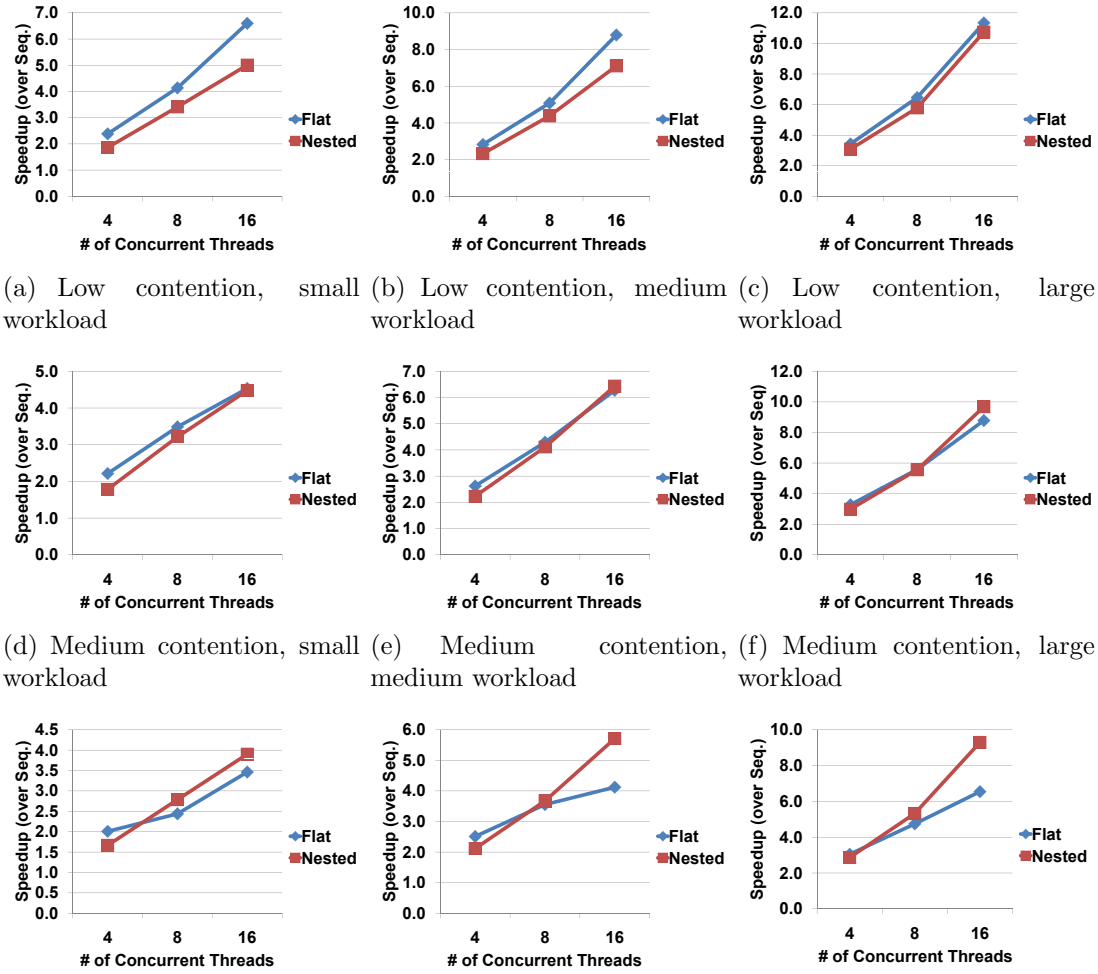
5.5.4 Q3: Improving Performance using Nested Parallelism

np-rbtree operates on a data structure that consists of multiple red-black trees. It performs two types of operations on the data structure: *look-up* operations that look up (read) entries and *insert* operations that insert (read/write) entries in the red-black trees. Insert operations often modify the trees in a global manner, causing many other transactions to abort. The ratio of look-up to insert operations is configurable. Each operation atomically accesses all the trees in the data structure using a transaction. After accessing each tree, np-rbtree executes the computational workload whose size is also configurable.

We exploit the parallelism in np-rbtree in two ways – (1) flat: parallelism at the outer level (i.e., inter-operation) and (2) nested: parallelism at both levels (i.e., inter-operation and inter-tree). If the percentage of the insert operations is high, the scalability of the flat version can be limited due to the frequent conflicts among top-level transactions.

We performed experiments by varying the two configurable parameters: the degree of contention and the size of the computational workload. Low, medium, and high contention cases are the ones where 1%, 5%, and 10% of the operations are inserts. Small, medium, and large workload cases are the ones where the computational workload iterates over 32, 128, and 512 loop iterations, respectively. In addition, np-rbtree performs 1024 operations, each accessing 8 red-black trees atomically.

Figure 5.15 shows that the flat version significantly outperforms the nested version with low contention and small workload. This performance difference results from the sufficient top-level parallelism (i.e., low contention) effectively exploited by the flat version and the unamortized overheads (i.e., small workload) of the nested version such as the runtime overheads of nested transactions. In contrast, the nested version greatly outperforms the flat version with high contention and large workload. This is because the nested version can effectively exploit the parallelism available at both levels and its overheads are sufficiently amortized using large workload. On the other hand, the scalability of the flat version is mainly limited due to the frequent conflicts among top-level transactions. The results also motivate future work on a nesting-aware TM runtime system that dynamically exploits the parallelism available in different



(a) Low contention, small workload

(b) Low contention, medium workload

(c) Low contention, large workload

(d) Medium contention, small workload

(e) Medium contention, medium workload

(f) Medium contention, large workload

(g) High contention, small workload

(h) High contention, medium workload

(i) High contention, large workload

Figure 5.15: Scalability of np-rbtree.

nesting levels based on runtime information such as the workload size and the degree of contention.

We also exploit nested parallelism to improve the overall performance of labyrinth in the STAMP benchmark suite [25]. This application implements a parallel version of Lee's routing algorithm using memory transactions [62, 100]. A three-dimensional grid is used to represent the maze. Each thread is assigned with pairs of start and end points in the grid. It connects a start point and an end point by a path of adjacent grid points that do not overlap with any other paths.

Each thread computes a path and adds it to the global grid using a single transaction. To reduce the degree of contention, this application uses a *privatization technique* [100]. A per-thread local copy of the global grid is created and used to compute the path. Each thread then attempts to add the computed path to the global grid by validating all the grid points in the path. If there is any invalid grid point (i.e., simultaneously accessed by another transaction) in the computed path, the transaction should abort and repeat the process by creating a new (updated) copy of the global grid. If the validation is successful, the computed path is added to the global grid atomically. Most of the execution time of this application is spent on the path computation [25].

Nested parallelism can be effectively used to improve the overall performance of labyrinth for the following reasons. First, labyrinth has very long transactions because each transaction exhaustively explores the grid using the breadth-first search (BFS) algorithm. Since the BFS algorithm is highly parallelizable, nested parallelism within top-level transactions can be effectively used to reduce their execution time. Second, with a larger number of concurrent top-level transactions, more conflicts can occur because multiple transactions are more likely to pick up overlapping paths. The increased contention will eventually limit the scalability of the application. Instead, using some of the available processors to exploit nested parallelism may lead to better performance.

To exploit nested parallelism, we implemented two versions of np-labyrinth. The *baseline* version of np-labyrinth uses nested parallel transactions to parallelize the BFS performed to compute paths. The *optimized* version of np-labyrinth is similar to its

baseline version except that it uses a *customized commit barrier*. With this customized commit barrier, a nested transaction does not merge its read and write sets to its parent when it commits. The key observation here is that nested transactions only access the local grid of their parent transaction. In other words, no other top-level transactions can access the same memory objects with the nested transactions of a particular top-level transaction. Therefore, top-level transactions do not need to track the read and write sets of their (committed) nested transactions, which allows for the use of the customized commit barrier.

In comparison with labyrinth, the baseline version of np-labyrinth introduces the following performance issues. First, the single-threaded execution time of the parallel BFS in np-labyrinth is increased due to the use of transactional barriers to mark the boundaries of nested transactions and synchronize concurrent accesses to the grid points by nested transactions. Note that no transactional barriers are required in labyrinth as it performs the sequential BFS. Second, since committing nested transactions merge their read and write sets, the read and write signatures of their parent transactions can be quickly saturated if the local grid is sufficiently large. The saturated read and write signatures may trigger more false conflicts due to aliasing. While still having the first performance issue (i.e., the runtime overheads of transactional barriers), the optimized version of np-labyrinth eliminates the second performance issue using the customized commit barrier.

Figure 5.16 shows the speedups of labyrinth (flat) and the two versions of np-labyrinth. The baseline version of np-labyrinth is outperformed by the other two versions at all thread counts due to the aforementioned runtime overheads. With a small number of threads, labyrinth significantly outperforms the optimized version of np-labyrinth due to the lower runtime overheads of single-level parallelism. With a large number of threads, labyrinth scales only marginally because of more frequent conflicts among the top-level transactions. In contrast, the optimized version of np-labyrinth continues to scale by effectively exploiting the parallelism available at both levels.

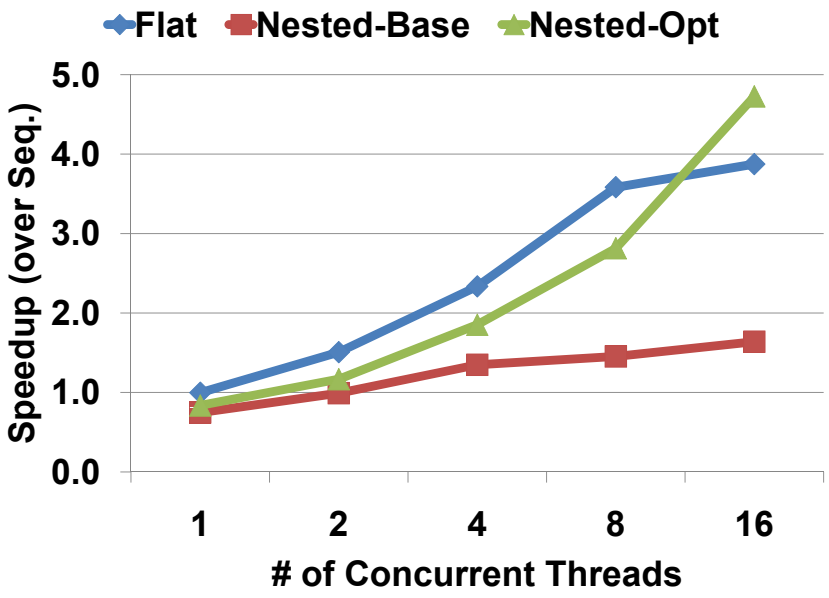


Figure 5.16: Scalability of labyrinth and np-labyrinth.

5.6 Related Work

Moss and Hosking proposed the reference model for concurrent nesting in TM [72]. Based on the proposed model, a few recent papers investigated nested parallelism in STM systems [8, 12, 15, 80, 96]. While compatible with existing multicore chips, this software-only approach may introduce excessive runtime overheads due to the use of complicated data structures [8, 15] or the use of an algorithm whose time complexity is proportional to the nesting depth [12], limiting its practicality. Our work differs because FaNTM aims to eliminate substantial runtime overheads of software nested transactions using hardware acceleration.

Vachharajani proposed an HTM system that supports nested parallelism within transactions [93]. While insightful, the proposed design drastically increases hardware complexity by intrusively modifying the hardware caches to implement nesting-aware conflict detection and data versioning. In contrast, FaNTM simplifies hardware by decoupling nesting-aware transactional functionality from the caches using hardware filters.

5.7 Conclusions

This chapter presented FaNTM, a hybrid TM system that provides practical support for nested parallel transactions using hardware filters. FaNTM effectively eliminates the excessive runtime overheads of software nested transactions using lightweight hardware support. FaNTM simplifies hardware by decoupling nested parallel transactions from the hardware caches. Through our performance evaluation, we demonstrated that FaNTM incurs a small runtime overhead when only single-level parallelism is used. We also showed that nested transactions on FaNTM perform comparably with top-level transactions and run significantly faster than those on NesTM. Finally, we showed how nested parallelism can be used to improve the overall performance of transactional applications.

Chapter 6

Implementing a Model Checker for TM

6.1 Introduction

Transactional Memory (TM) takes responsibility for managing all accesses to shared state, thus extreme attention must be paid to its performance. Consequently, subtle but fast implementations are favored over simpler ones, even though it makes the resulting implementations difficult to prove correct. TM's central position, however, means that the severity of any bug is magnified.

To address this problem, a few recent proposals have attempted to formally verify the correctness of TM systems using model checking techniques. In [34], Cohen et al. proposed a formal method to verify the correctness of the TM systems similar to TCC [46] and LogTM [69] using the TLA+ model checker [60]. In [44], Guerraoui et al. proved an important reduction theorem that states the verification problem can be reduced to the most general problem with two threads and two shared variables if an evaluated TM system satisfies a set of certain conditions. They then verified the correctness of the abstract models of several software TMs (STMs) such as TL2 [37] and DSTM [52]. Finally, O'Leary et al. [75] verified the correctness of Intel's McRT STM [83] using the Spin model checker [54].

However, there still remain important research issues that require further investigations. First, TM systems should be modeled close to the implementation level in order to reveal as many potential bugs as possible. For instance, the TL2 model in [44] does not model the timestamp-based version control mechanism, which requires a hand proof that their abstract model is equivalent to the actual implementation. Second, model checking should be extended to a wide range of TM systems that use additional hardware components (e.g., hybrid TM systems) or support nested parallelism. Third, both transactional and non-transactional memory operations should be modeled to investigate subtle correctness issues with weak isolation and ordering [86]. Finally, an in-depth, quantitative analysis is required to understand practical issues such as the sensitivity of the state space to various system parameters and to motivate further research in model checking TM systems.

This chapter presents *ChkTM*, a flexible model checking environment for checking the correctness of TM systems. ChkTM aims to model TM systems close to the implementation level. For example, the version control mechanism of timestamp-based STM systems can be accurately modeled in ChkTM using our *timestamp canonicalization* technique. In addition, ChkTM can flexibly model TM systems that use additional hardware components or support nested parallelism. Furthermore, both transactional and non-transactional memory operations are also modeled in ChkTM.

The specific contributions of this work are:

- We propose a flexible model checking environment for TM (ChkTM) that can be used to check the correctness of various TM systems. ChkTM consists of the three components: (1) an architectural state space explorer, (2) TM model specifications, and (3) a test program generator.

- In ChkTM, we model several TM systems including a widely-used high-performance STM (TL2) [37], a hybrid TM (SigTM) [26] that accelerates an STM using hardware signatures, and an STM (NesTM) that supports nested parallel transactions [12].

- We introduce the timestamp canonicalization technique that accurately models the version control mechanism used in TL2 and NesTM.

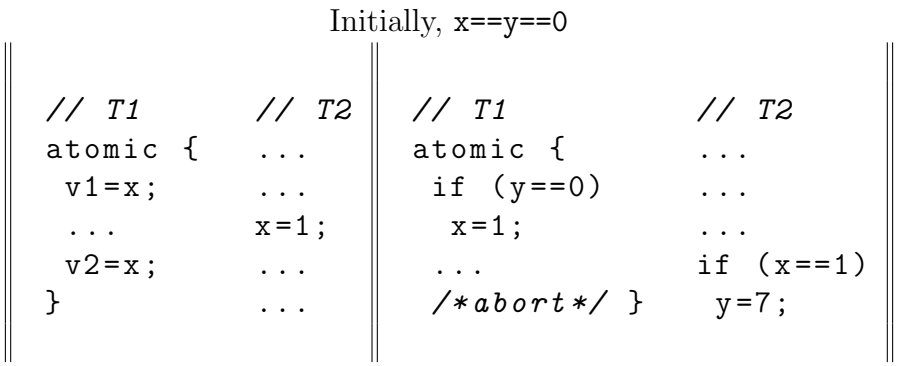
- We describe a case study in which we actually found a subtle correctness bug in the current implementation of eager-versioning TL2. We reported this bug to the TL2 developers.

- Using ChkTM, we check that TL2 and SigTM guarantee the serializability of every possible execution of every possible transactional program with two threads, each executing one transaction that performs at most three transactional memory operations. We also check that SigTM provides strong isolation for the test programs described in [86].

- We perform an in-depth, quantitative analysis on ChkTM to understand the practical issues in model checking TM systems. First, we investigate the sensitivity of the state space to system parameters such as the number of concurrent threads. Second, we study the scalability of the multi-threaded ChkTM to reduce the latency of the verification. Third, we investigate the tradeoff between the performance and correctness of the verification when various approximation techniques are applied to the TM model. Our quantitative analysis also motivates further research on a reduction theorem [44] and dynamic partial order reduction techniques [43] for verifying TM systems with support for nested parallelism.

The rest of this chapter[1] is organized as follows. Section 6.2 provides background information. Section 6.3 describes the design and implementation of ChkTM. Section 6.4 presents the main correctness results and an in-depth, quantitative analysis on ChkTM. Section 6.5 reviews related work and Section 6.6 concludes this chapter.

[1]The work presented in this chapter was also published in [11].



(a) Non-repeatable reads (b) Speculative dirty reads

Figure 6.1: Violations of strong isolation.

6.2 Background

Correctness Criteria for TM

Serializability: The main correctness criterion used in this chapter is *conflict serializability* [88]. We will discuss how ChkTM checks the serializability of every possible execution of TM programs in Section 6.3.1.

Strong isolation: A TM system provides *strong isolation* if transactions are isolated from other transactions and non-transactional memory accesses [61]. Implementing strong isolation in STM systems has an unattractive performance impact on all non-transactional code. While performance optimization techniques have been proposed using whole-program static analysis [86] or dynamic recompilation techniques [24, 85], few STMs provide strong isolation.

In contrast, hardware TMs (HTMs) and some hybrid TMs [16, 26] (including SigTM investigated in this chapter) provide strong isolation with low overheads based on hardware support. Some STMs provide partial isolation guarantees that can be implemented more efficiently than strong isolation [35]. Among the TM systems we examine in this chapter, only SigTM provides any isolation for non-transactional accesses.

Figure 6.1 shows two violations of strong isolation, as discussed in [86]. Figure 6.1(a) illustrates a *non-repeatable read* (NR). The expected program behavior is

that T1's two reads to `x` should observe the same value. However, since T2 updates `x` without using a write barrier, T1 cannot detect the conflict on `x`. Hence, the two reads to `x` may return different values. Both lazy and eager STMs are vulnerable to NR.

Figure 6.1(b) shows a *speculative dirty read* (SDR) where a non-transactional read may observe a value speculatively written by a transaction. The expected program outcome is `x==1`. However, since eager STM speculatively updates shared memory in place on write (i.e., `x=1`), T2 may observe the value speculatively written by T1 and set `y` to 7. When T1 re-executes the transaction after it aborts, it will not update `x` because the value of `y` has been already updated to 7 by T2. In contrast, lazy STM is invulnerable to SDR as the updates made by a transaction are invisible (i.e., buffered) outside the transaction until it commits. We refer readers to [86] for an in-depth discussion on weak isolation behaviors in lazy and eager STMs.

Evaluated TM Systems

This chapter investigates the correctness issues of eager and lazy versioning TL2 software TM systems [25, 37, 38], NesTM [12], and eager and lazy versioning SigTM hybrid TM systems [25, 26]. Refer to Section 2.4 for additional details on eager and lazy versioning TL2 and SigTM systems. In addition, Chapter 4 provides a detailed discussion on NesTM.

6.3 Design and Implementation of ChkTM

Figure 6.2 presents the overall architecture of ChkTM that consists of the following three components: (1) an *architectural state space explorer* that exhaustively explores every reachable architectural state, (2) *TM model specifications* where the target TM systems are modeled close to the implementation level, and (3) a *test program generator* that generates a set of small TM programs. All the components are implemented in the *Scala* programming language [3]. We decided to use the Scala programming language because its domain-specific language features allowed us to implement ChkTM

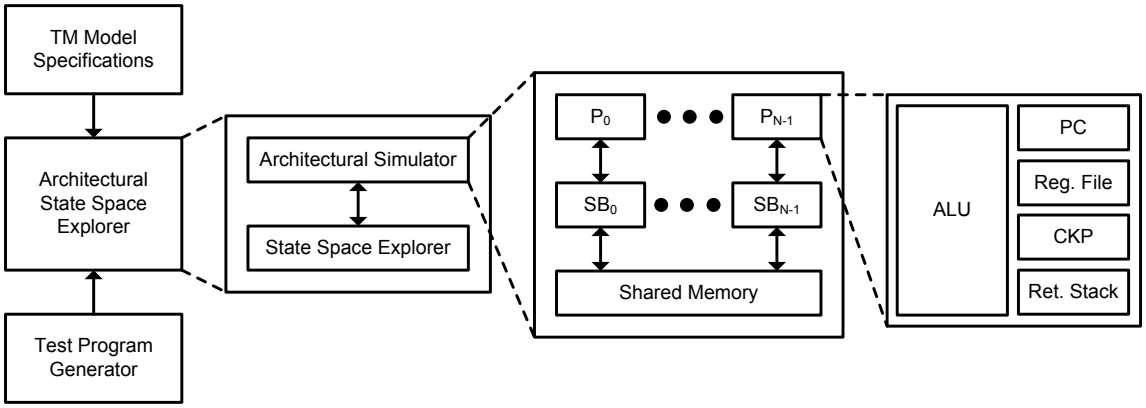


Figure 6.2: The overall architecture of ChkTM.

in a concise manner. We will discuss the three components of ChkTM in a bottom-up approach.

6.3.1 Architectural State Space Explorer

As illustrated in Figure 6.2, the architectural state space explorer (ASE) consists of two main components: (1) an architectural simulator and (2) a state space explorer. The *architectural simulator* models the internal architectural state of a simple shared-memory multiprocessor system that consists of processors, store buffers, and shared global memory. Each processor in the simulator models a RISC processor with an ALU, a program counter, a register file, a register checkpoint, and a return stack. Every update to shared memory is made via a bounded store buffer (SB) that may retire stores in any order, which provides a memory consistency model similar to SPARC's Total Store Order (TSO) [89]. An explicit memory fence instruction and a compare-and-swap instruction cause SB flushes. If the SB size is set to 0, the simulator models sequential consistency. While we use only sequential consistency in this work, verifying TM systems under relaxed consistency models is interesting future work. Finally, all the system parameters such as the number of processors and shared memory size are configurable.

The second component of ASE is the *state space explorer*. In the state space explorer, states are represented as persistent trees. State transitions are pure functions

```
1: procedure EXPLORE
2:   frontiers ← InitialState
3:   while frontiers ≠ ∅ do
4:     nextFrontiers ← ∅
5:     for all currState in frontiers do
6:       hasSucc ← false
7:       for all trans in AllPossibleTransitions do
8:         hasSucc ← true
9:         nextState ← trans(currState)
10:        transPair ← (currState,nextState)
11:        if hist.contains(transPair)=false then
12:          hist.put(transPair)
13:          nextFrontiers.put(nextState)
14:      if hasSucc = false then
15:        terminals.put(currState)
16:    frontiers ← nextFrontiers
```

Figure 6.3: Pseudocode for the state space explorer.

that produce a new state without altering the old. Figure 6.3 illustrates how the state space explorer performs a breadth-first search (BFS)[2] to determine all the possible final outcomes when executing a small program using a TM model.

To reduce the state space, we implemented a simple optimization where instructions are merged with their successor if the instruction's dynamic execution accesses only processor-private values. Since these intermediate values do not affect the outcome, this optimization does not affect the correctness of model checking.

To check the serializability of a TM system, ChkTM first performs the *coarse-grain state space exploration* (CSE) where only a single processor is active at any time and where the active processor cannot be changed while a transaction is active. CSE directly enumerates all serial schedules. The terminal states produced by CSE are used as valid terminal states. We augment the architectural state to record the values observed by transactional reads (VOR) and the values overwritten by transactional

[2]Since the state space explorer is based on BFS, it is highly parallelizable. To implement the multi-threaded state space explorer, we parallelized the main loop (i.e., line 5 in Figure 6.3) by dynamically assigning a chunk of iterations to each thread at a time. We study its scalability in Section 6.4.

Initially, x==y==0

```
// T1          // T2
atomic {       atomic {
 ld x           st x,200
 st y,101 }     ld y }

VOR(T1)={(x,0)}
VOR(T2)={(y,101)}
x=200, y=101

VOR(T1)={(x,200)}
VOR(T2)={(y,0)}
x=200, y=101
```

(a) Test code and valid results

```
T1: ld x
T2: st x,200
T2: ld y
T1: st y,101

VOR(T1)={(x,0)}
VOR(T2)={(y,0)}
x=200, y=101
```

(b) An invalid execution

Figure 6.4: Detecting a serializability violation using the values observed by transactional reads (VOR).

writes (VOW). VOR and VOW are retained only for committed transactions. In addition, ChkTM ensures that every store in a test program writes a unique value. This establishes one-to-one mapping between values recorded in VOR and VOW, and the conflicting read or write operations. After CSE is performed, ChkTM records the VOR, VOW, and final memory state corresponding to every serial execution. These are the valid terminal states.

Next, ChkTM performs the *fine-grain state space exploration* (FSE) which explores every possible memory access interleaving. When a terminal state is reached during FSE, ChkTM checks that the VOR, VOW, and final memory state of the terminal state are identical to one of the valid terminal states produced by CSE. Figure 6.4 illustrates how ChkTM detects a serializability violation using the VOR and the final memory state. Figure 6.4(a) shows a simple test program and a set of valid terminal states produced by CSE. Figure 6.4(b) illustrates an invalid execution trace and an invalid terminal state produced during FSE. ChkTM reports a serializability

Initially, x==0

```
// T1          // T2
atomic {       atomic {
 st x,100       st x,200 }
 st x,101 }

VOW(T1)={(x,0),(x,100)}
VOW(T2)={(x,101)}
x=200

VOW(T1)={(x,200),(x,100)}
VOW(T2)={(x,0)}
x=101
```

(a) Test code and valid results

```
T1: st x,100
T2: st x,200
T1: st x,101

VOW(T1)={(x,0),(x,200)}
VOW(T2)={(x,100)}
x=101
```

(b) An invalid execution

Figure 6.5: Detecting a conflict-serializability violation using the values overwritten by transactional writes (VOW).

violation by detecting the terminal state reached during FSE does not match to any valid terminal state in Figure 6.4(a).

A serializability test using only the VOR and final memory state checks view serializability. To check conflict serializability, we add VOW. The key observation behind using VOW is that any view serializable, but not conflict serializable schedule contains a blind write [88]. VOW allows us to check that the ordering of all the write operations of a terminal state reached during FSE is identical to the one produced by a serial schedule in CSE. Figure 6.5 demonstrates an example of how ChkTM detects a conflict serializability violation (but still view serializable) of an execution using VOW.

Finally, similar to checking serializability, ChkTM checks strong isolation as follows. Given a test program, ChkTM produces all the possible expected outcomes discussed in Section 6.2 by performing CSE. Then, ChkTM performs FSE to explore every possible execution of the test program. If any unexpected outcome is detected during FSE, ChkTM reports a strong isolation violation.

```
long TxLoad(Self, addr)
{
 if(Self.WS.contains(addr)){




  val=Self.WS.lookup(addr);


  Self.RS.insert(addr);


  return val; }
 cv = getVo(addr);

 val = *addr;

 if(isLocked(cv) ||
    extractTS(cv)>Self.rv ||
    cv!=getVo(addr)){
  TxAbort(Self);
 }

 Self.RS.insert(addr);


 return val;
}
```

(a) Pseudocode

```
// TxLoad model
// BX: addr, AX: return value
instrs = instrs ++ (List(
 (0 -> new Instr {
  nextPC=TxLoad+
  (if (ws.contains(bx))
    10 else 40) }),
 (10 -> new Instr {
  assign(AXKey(cpu),
  ws.apply(bx).wsVal) }),
 (20 -> new Instr {
  rs = rs + bx
  reads = (bx, ax) :: reads}),
 (30 -> new Ret),
 (40 -> new Instr {
  vx = read(VoLockKeys(bx))}),
 (50 -> new Instr {
  ax = read(AppMemKeys(bx))}),
 (60 -> new Instr {
  nextPC =
  (if ((extractOwn(vx)!=-1) ||
  (extractTS(vx)>rv) || (
   vx!=read(VoLockKeys(bx))))
   TxAbort else (TxLoad+70))}),
 (70 -> new Instr {
  rs = rs + bx
  reads = (bx, ax) :: reads}),
 (80 -> new Ret),
).map(e=>(e._1+TxLoad,e._2)))
```

(b) Modeled code

Figure 6.6: Comparison between the C-language styled pseudocode and the Scala-language styled model in ChkTM for `TxLoad` in lazy TL2.

6.3.2 TM Model Specifications

Modeling TL2

To model TL2, the baseline ASE is augmented with additional state variables including a global version clock, voLocks, read and write sets of transactions. The global version clock and voLocks are modeled as globally-visible variables because they are accessed by multiple processors in the system. In contrast, the read and write sets of each transaction are modeled as processor-private variables.

Then, the algorithms of lazy and eager TL2 are specified using the Scala programming language. Figure 6.6 compares the C language-styled pseudocode and the model specified in Scala for `TxLoad` in lazy TL2. Figure 6.6 clearly shows that ChkTM aims to model an evaluated TM system at the implementation level to reveal as many potential bugs as possible. Other TM barriers are also modeled similarly.

As noted in [75], it is challenging to accurately model timestamp-based STMs such as TL2. This is because an infinite number of states correspond to serializable executions with different timestamp values. To address the state space explosion problem, we propose the *timestamp canonicalization* technique. The key observation behind this technique is that the relative ordering among timestamp values is important to order transactional events, but the exact timestamp values are unimportant. Based on this observation, after each step, ChkTM canonicalizes all the timestamps that are present in the architectural state. There are three steps: (1) compute the set of the timestamp values present anywhere in the architectural state, (2) sort them, and (3) replace each value with its ordinal position in the sorted set. The canonicalized variables are the global version clock (GC), the voLocks, and any registers that are holding a timestamp value. Figure 6.7 shows an example of timestamp canonicalization. Note that the least significant digit (in decimal) of voLock and VX register values is not canonicalized because it encodes the ownership information.

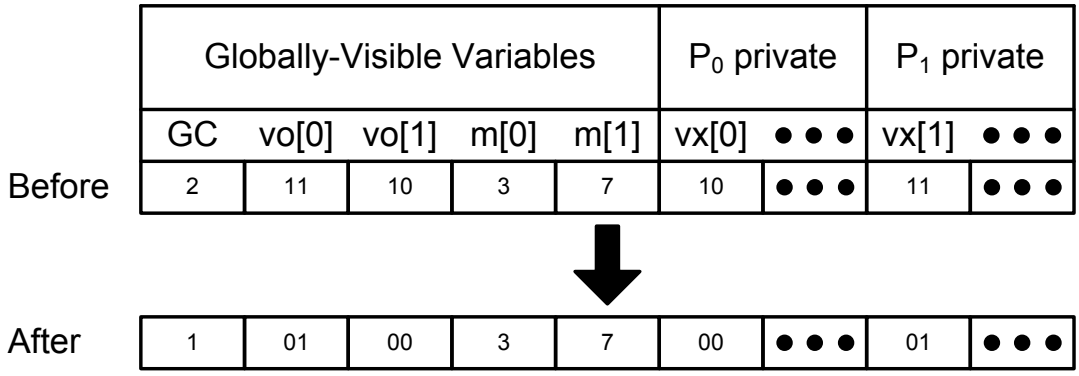


Figure 6.7: An example of timestamp canonicalization.

Modeling NesTM

Apart from the state variables introduced to model TL2, the NesTM model uses additional state variables including a doomed bit, a rollback counter, and a commit lock per each processor. Furthermore, additional instructions such as `fork` and `join` are modeled to initiate and terminate the execution of nested threads. Unlike (non-nested) TL2, the read and write sets of each processor are also modeled as globally visible variables because they can be accessed by child transactions running on different processors [12]. Finally, since the rollback counter may also cause a state space explosion, ChkTM performs *rollback-counter canonicalization*, in addition to timestamp canonicalization.

Modeling SigTM

To model SigTM, the baseline ASE is augmented with additional state variables to represent per-processor hardware signatures (and their state information such as `lookUp` and `WSigNack`), and transactional state information such as a doomed bit. All of them are modeled as globally visible state variables because they can be accessed by other processors. Unlike TL2 and NesTM, timestamp variables are not used to model SigTM. Additional memory operations are also modeled such as `fetchExclusive` and non-transactional load and store instructions.

```
atomic {
 if (exec) {
  if (read) {
   TxLoad(genAddr);
  } else {
   TxStore(genAddr,genUniqVal);
  }
 }
 // repeat the process as necessary
 ...
}
```

Figure 6.8: The skeleton transaction code used for tests.

Currently, ChkTM models SigTM based on a zero-latency interconnection network. We expect that prior work on model checking cache coherence protocols [56] can be used to extend the current hardware model in ChkTM.

6.3.3 Test Program Generator

To check the serializability of TM systems, the test program generator of ChkTM uses the skeleton code shown in Figure 6.8. A transaction executes a specified number of transactional memory operations. For each memory operation, there are three parameters to consider: (1) whether it is performed or not, (2) the type of memory operation (i.e., read or write), and (3) the accessed memory location. All these parameters can be generated either randomly (for random tests) or systematically (for exhaustive, complete tests). Also note that the value written by each write operation is unique with respect to all the other written values. Finally, to check the strong isolation guarantees of TM systems, we manually coded the test programs described in [86].

Initially, `x==y==0`

```
// T1                  // T2
atomic {               atomic {
  st x,1                 st y,2
  ld y                   ld x
}                      }
```

Can VOR(T1)=={(y,2)} and VOR(T2)=={(x,1)}?

Figure 6.9: A simple test program used to test (buggy) eager TL2.

6.3.4 Case Study: Invalid-read Problem

To showcase the effectiveness of ChkTM, we present a case study in which we found the *invalid-read* bug in the available implementation of eager TL2. To check the correctness of eager TL2, we modeled it and performed exhaustive tests using ChkTM. During the tests, ChkTM reported a serializability violation with the test program in Figure 6.9 and generated invalid executions including the one shown in Figure 6.10.

To investigate the cause of the problem, we analyze the invalid execution in Figure 6.10. T1 executes the code in `TxLoad`, while T2 executes the code in `TxStore` and `TxAbort`. At step 0, T1 samples the current value of the voLock for `addr` which is the address of `y`. At steps 1 and 2, T2 acquires the voLock and writes a value of 2 to `addr` during the execution of `TxStore`. At step 3, T1 reads the speculatively written (i.e., 2) value from `addr`. At steps 4 and 5, during the execution of `TxAbort`, T2 restores the memory and voLock values for `addr` to the previously observed values. At step 6, the `if` statement tests whether the value of the voLock sampled at step 0 matches the current value of the voLock (i.e., `cv!=getVo(addr)`?). Since T2 has already restored the value of voLock to the previous value (i.e., the same value as `cv`), the test at step 6 fails (i.e., `cv==getVo(addr)`). Therefore, T1 assumes that the memory value read at step 3 is still valid and successfully commits in the end. This is an execution scenario where the invalid outcome in Figure 6.9 is produced.

By analyzing the invalid execution, we identified that there is a subtle correctness bug in line 5 of the code of `TxAbort` shown in Figure 6.11. To avoid the invalid-read

```
    // T1: TxLoad                  // T2: TxStore,TxAbort
0:  cv=getVo(addr)                 ...
1:  ...                            lock(addr)
2:  ...                            *addr=2
3:  val=*addr // *addr==2          ...
4:  ...                            *addr=0
5:  ...                            unlock(addr)
6:  if(...cv!=getVo(addr)){        ...
7:    ... }                        ...
```

Figure 6.10: An unserializable execution detected by ChkTM in (buggy) eager TL2.

```
1: procedure TxAbort
2:   for all addr in ws do
3:     Mem[addr] ← ws.lookup(addr)
4:   for all addr in ws do
5:     unlock(addr)              ▷ Timestamp value should have been incremented.
6:   rs.reset()
7:   ws.reset()
8:   runContentionManager()
9:   restoreCheckpoint()
```

Figure 6.11: A bug (in line 5) found in `TxAbort` in eager TL2.

problem, aborting transactions must increment the timestamp values of the acquired voLocks instead of merely restoring them to the previously observed values. We reported this bug to the TL2 developers.

Readers should also note that it may be very difficult to find this kind of subtle correctness bugs by performing random tests using the implemented TM code on real machines. One could increase the possibility of revealing such bugs from random tests by inserting randomized delays at various locations in the TM code (e.g., between the operations at steps 0 and 3 in Figure 6.10). However, it requires programmer's non-trivial intuitions on where potential bugs would be in the code. In contrast, ChkTM can reveal any potential bug without requiring programmer's intuitions because it explores every possible execution of a TM program, even unlikely ones.

6.3.5 Discussion

State space reduction: While we implemented a simple optimization that merges locally visible steps to reduce the state space in ChkTM, it is an interesting, open research question to investigate more aggressive optimizations, such as partial order reduction (POR) techniques [43]. As noted in [75], traditional POR techniques cannot be directly applied due to their conservativeness when iterations or global variables are used in the TM model. In particular, we note that it would be interesting to extend the dynamic POR techniques [40] in the context of model checking TM systems.

Modeling other TM systems: Because ChkTM addresses the state space explosion problem due to the use of timestamps (or any kind of counters) using the timestamp canonicalization technique, it is straightforward to model other STMs in ChkTM. To model other hybrid or hardware TM systems, ChkTM should be augmented with additional state variables to model the hardware components used in the evaluated TM systems. An open question is how to model asynchronous communications with hardware components in detail without causing a state space explosion.

Verifying liveness: ChkTM could be extended to verify the liveness of TM systems by detecting cycles in the state transition graph. The existence of a cycle indicates that there is an execution scenario in which a TM program does not terminate (e.g., transactions infinitely abort and restart). Note that all the evaluated TM systems in this chapter are known to admit livelock, but attempt to probabilistically provide liveness using randomized backoff schemes.

6.4 Evaluation

In this section, we first discuss our main correctness results for TL2 and SigTM, and describe our progress in model checking NesTM. We then present an in-depth, quantitative analysis on ChkTM to understand the practical issues in model checking TM systems. More specifically, we investigate (1) the sensitivity of the execution time and the number of the explored states to the parameters such as the test program size and the number of threads, (2) the scalability of the multi-threaded ChkTM, and

TM System	Total time (s)	Avg. time (s)	Avg. # of states
TL2-L	7971	0.51	2115
TL2-E	13453	0.98	10712
SigTM-L	10494	0.67	4931
SigTM-E	8227	0.53	2279

Table 6.1: Total, average execution time, and average number of the explored states required to check the serializability of TM systems.

(3) the tradeoff between the performance and correctness of the verification when approximation techniques are applied to the TM model.

We performed experiments on computers with two quad-core 2.33GHz Intel Xeon CPUs and 32GB of shared memory. We used Linux `x86_64` kernel 2.6.18 and the 64-bit Server VM in Sun's Java^{TM}SE Runtime Environment, build 1.6.0-14-b08. We used an 8GB heap for model checking TL2 and SigTM and a 30GB heap for model checking NesTM. Finally, we used the Scala compiler (version 2.7.5) to compile the ChkTM code.

6.4.1 Correctness Results

Serializability: To check the serializability of TL2 and SigTM, we first generated all the possible test programs that satisfy the following conditions[3]: (1) two threads in each program, (2) one transaction per thread, (3) at most three transactional memory operations (i.e., read or write) per transaction, (4) no non-transactional memory operations in the program (i.e., purely transactional). In addition, we assumed that the underlying system has two shared-memory words and provides sequential consistency. We then ran all the generated test programs on the TL2 and SigTM models in ChkTM and checked that no errors were reported.

Table 6.1 summarizes the execution time and the number of the explored states required to check each TM system. It takes less than a second to run each test and 2–4 hours to exhaustively check each TM system with all the possible test programs. Table 6.1 shows that the explored state space in checking eager TL2 is significantly

[3]The use of this configuration was inspired by the approach discussed in [75].

WI Anomaly	TL2-L	TL2-E	SigTM-L	SigTM-E
NR	Y	Y	N	N
ILU	Y	Y	N	N
IDR	N	Y	N	N
SLU	N	Y	N	N
SDR	N	Y	N	N
OW	Y	N	N	N
BW	Y	N	N	N

Table 6.2: Summary of the weak isolation (WI) anomalies in TL2 and SigTM.

larger than the other TM systems. Since the timestamp values are incremented even when transactions abort to avoid the invalid-read problem, more unique states associated with the incremented timestamp values are generated and explored in checking eager TL2.

Strong isolation: To investigate the strong isolation guarantees of TL2 and SigTM, we manually coded the test programs described in [86]. We tested seven of the nine anomalies, including non-repeatable reads (NR), intermediate lost updates (ILU), intermediate dirty reads (IDR), speculative lost updates (SLU), speculative dirty reads (SDR), overlapped writes (OW), and buffered writes (BW). We omitted the granularity-related anomalies because they can only be solved at a level higher than a word-based STM. Table 6.2 summarizes the results. For every possible execution of all the test programs, SigTM (both lazy and eager) did not produce any unexpected final outcome. In contrast, ChkTM successfully detected weak isolation anomalies for TL2. In summary, we were able to mechanically check the arguments in [86] based on the memory operations (both transactional and non-transactional) modeled in ChkTM.

6.4.2 Quantitative Analysis

Sensitivity to the system parameters: We investigate the sensitivity of the execution time and the number of the explored states in checking TM to various system parameters. The parameters on which we focus are the program and shared memory sizes and the number of threads. Figure 6.12 shows the results for the program and

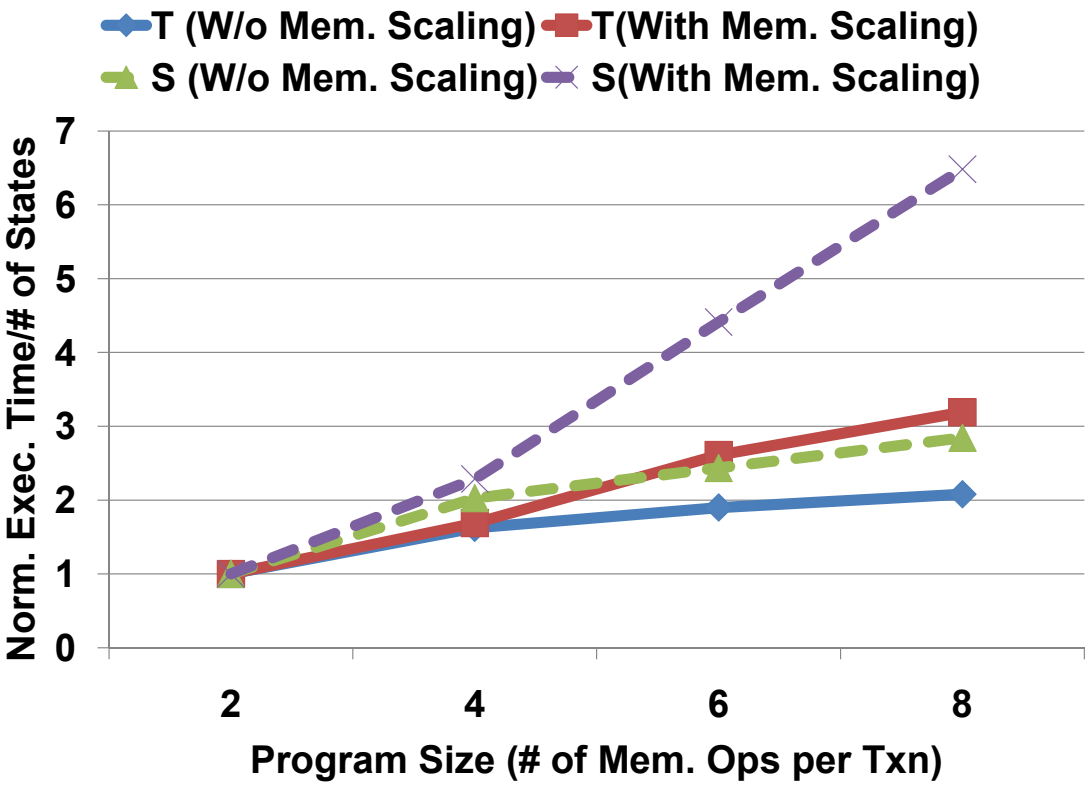


Figure 6.12: Sensitivity of the execution time and the number of the explored states to the test program and shared memory sizes.

shared memory sizes. The program size represents the number of memory operations performed by each transaction. The shared memory size indicates the number of shared-memory words in the system. The first result data set is collected by just scaling the program size while the shared memory size is fixed at two (i.e., two shared memory words in the system). In contrast, the second result data set is collected by incrementing the shared memory size by 1 when the program size is incremented by 2. In the baseline configuration, program and shared memory sizes are set to 2. All the results are normalized to the baseline result. Finally, the result with each configuration is collected by running 100 randomly generated test programs on the lazy TL2 model.

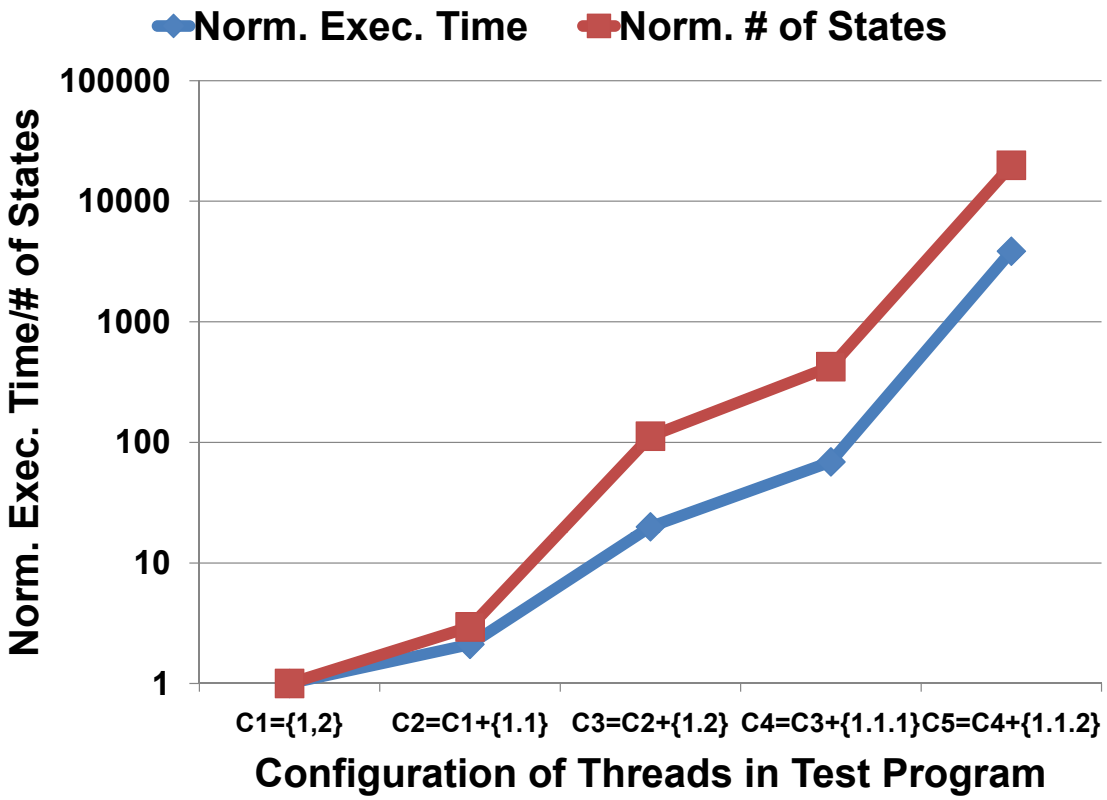


Figure 6.13: Sensitivity of the execution time and the number of the explored states to the number of threads.

Figure 6.12 shows that the execution time and the number of states increase rather slowly when only the program size increases. With a small shared memory size, each operation conflicts with other operations with a high probability. This limits the number of possible interleavings as each operation has dependencies with other operations due to the conflicts. Hence, the state space grows rather marginally even when each transaction performs more operations. In contrast, with a larger shared memory size, each operation can execute more independently with respect to the other operations due to lower contention. Therefore, more interleavings can be produced and the state space grows much faster when both program and shared memory sizes increase.

We also investigate the sensitivity of the execution time and number of states to the number and configuration of threads when checking NesTM. Figure 6.13 presents the sensitivity results to different number and configuration of threads. Specifically, C1 is the configuration in which two top-level transactions (i.e., T1 and T2) run. C2 is generated by adding T1.1 (i.e., a child of T1) to C1. C3 is generated by adding T1.2 (i.e., a child of T1 and sibling of T1.1) to C2. C4 and C5 are repeatedly generated in a similar manner. Each transaction executes only two reads, to reduce the state space.

Figure 6.13 shows that the execution time and the number of states required to check NesTM increases explosively when the number of threads increases (note that y-axis in Figure 6.13 has a log scale). One interesting observation is that the increase is much larger when a new sibling thread is added (e.g., from C2 to C3, from C4 to C5). This is due to the fact that there is no ordering dependency between sibling transactions, thus more interleavings can be produced. In contrast, when a new (single) child thread is added (e.g., from C1 to C2, from C3 to C4), the increase is relatively smaller. This is because the newly added child has an ordering dependency with its parent, thus the number of possible interleavings is limited. For example, T1.1 can run only after T1 forks it, and T1 is suspended while T1.1 is active.

The result in Figure 6.13 clearly motivates the need for a reduction theorem applicable to the TM systems with support for nested parallelism. While two threads and two variables are proved sufficient for non-nested TM systems [44], no such guarantee of completeness is available for our checks of NesTM. Since the state space explosively grows with the increasing number of threads and nesting levels, model checking may be feasible only for small configurations. For example, when larger test programs are used with the thread configuration C5, it is currently impossible to check NesTM using ChkTM even on computers with 32GB physical memory due to a state space explosion. Finally, dynamic state space reduction techniques should be also investigated to make it feasible to check the nested TM systems on commodity machines.

Scalability of the multi-threaded ChkTM: To investigate the scalability of the multi-threaded ChkTM, we use a test program with the thread configuration C5 on

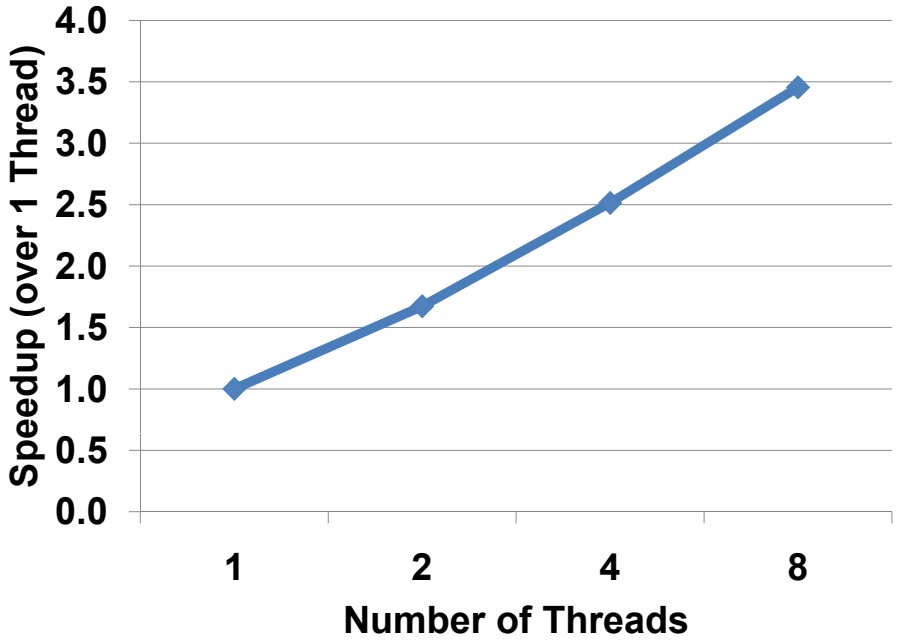


Figure 6.14: Speedup of the multi-threaded ChkTM.

the NesTM model. To avoid a state space explosion, each transaction in the test program performs only two read operations. We varied the number of concurrent threads from 1 to 8, and measured the speedup by dividing the execution time with multiple threads by the one with 1 thread. As shown in Figure 6.14, the multi-threaded ChkTM shows reasonable scalability when checking NesTM (e.g., speedup of 3.5× with 8 threads).

Tradeoff between performance and correctness: One commonly-used technique to reduce the state space is to apply approximations by merging several steps of the modeled algorithm into one. Obviously, a careless use of such approximations may affect the correctness of the verification. To investigate the tradeoff between the performance and correctness of the verification, we apply a set of approximations when checking NesTM. Table 6.3 summarizes the approximations. For example, Approximation 1 merges the steps of reading the voLock and the value of a memory object (i.e., the operations at step 0 and 3 in Figure 6.10 are assumed to happen atomically).

Approx.	Barrier	Description
#1	L	Atomically read voLock and mem. value
#2	C	Atomically release voLocks
#3	A	Atomically release voLocks
#4	S	Atomically insert a WS entry and write mem. value
#5	A	Atomically rollback mem. values
#6	C,A,S	All except for Approx. 1

Table 6.3: Approximations applied to the NesTM model. L, S, C, and A indicate `TxLoad`, `TxStore`, `TxCommit`, and `TxAbort`.

Version	Exec. Time (s)	Num. of States	Fidelity
Base	11.70	619,864	Y
Approx. 1	10.51	533,996	N
Approx. 2	8.82	415,530	Y
Approx. 3	11.79	617,851	Y
Approx. 4	11.46	602,791	Y
Approx. 5	11.52	618,011	Y
Approx. 6	8.29	399,058	Y

Table 6.4: Average execution time, number of explored states, and the fidelity of the baseline and approximated NesTM models.

For another example, Approximation 2 releases all the acquired voLocks on commit in a single step (i.e., the entire loop is atomically executed).

Table 6.4 summarizes the average execution time and the number of explored states for checking the baseline and approximated NesTM models. The results were collected using 10 randomly generated test programs in which the thread configuration C3 is used and each transaction executes at most two memory operations. Table 6.4 shows that Approximation 2 is effective in reducing the state space. On commit, a transaction tends to have more entries in its write set because it has executed all of its memory operations. Hence, Approximation 2 can effectively reduce the state space by merging more steps in the algorithm. In contrast, approximations applied to the abort barrier are less effective. On abort, a transaction tends to have less entries in its write set because it often fails to execute all of its memory operations. Therefore, the approximations applied to the abort barrier are less effective because

they merge fewer steps. In summary, Approximation 6 (all the approximations except for Approximation 1) reduces the execution time and the number of states by 29.1% and 35.6%, respectively.

Table 6.4 also summarizes the fidelity of the baseline and approximated NesTM models when the invalid-read bug discussed in Section 6.3.4 is intentionally injected. With the injected invalid-read bug, the model with Approximation 1 does not report an error (i.e., false negative). This is because merging the aforementioned steps makes it impossible to produce any invalid execution similar to the one shown in Figure 6.10. In contrast, other approximated models do not show any false negative (at least for the invalid-read bug with the test programs we used). If a certain approximation can be proven that it does not affect the correctness of model checking, it can be used as an effective tool to reduce the state space.

6.5 Related Work

While there is a large amount of previous work on TM, relatively little work has been done on formally verifying TM systems. Cohen et al. [34] proposed a formal method to verify the correctness of a few TM systems similar to TCC [46] and LogTM [69] using the TLA+ model checker [60]. Guerraoui et al. [44] proved an important reduction theorem which states that the TM verification problem can be reduced to the most general problem with two threads and two shared variables, when an evaluated TM system satisfies a set of certain conditions. In addition, they verified the correctness of the abstract models of several STM systems such as DSTM and TL2. While insightful, these prior theoretical proposals modeled the evaluated TM systems rather abstractly. For instance, the TL2 model in [44] does not model the timestamp-based version control mechanism, which requires a hand proof that their abstract model is equivalent to the actual implementation. In contrast, ChkTM aims to model the evaluated TM systems close to the implementation level, to find as many potential bugs as possible.

In [75], O'Leary et al. verified the correctness of Intel's McRT STM using the Spin model checking environment [54]. Our work is similar to theirs in the sense that both

proposals attempt to model TM systems close to the implementation level. However, our work differs in the following aspects. First, we extend the use cases of model checking TM by investigating a wider range of TM systems, including an industrial, high-performance STM (TL2), a hybrid TM (SigTM) that uses hardware signatures, and an STM (NesTM) with support for concurrent nesting. Second, ChkTM models both transactional and non-transactional memory operations to enable studies on subtle correctness issues with weak isolation and ordering. Finally, we provide an in-depth, quantitative analysis on ChkTM to investigate the practical issues and motivate further research in model checking TM systems.

Finally, Manovit et al. proposed an axiomatic formulation to model the formal specification of a TM system and used it with random testing to find potential bugs in the evaluated TM system [64]. Our work differs in the sense that ChkTM aims to formally check the correctness of TM systems by exploring all possible executions.

6.6 Conclusions

This chapter presented ChkTM, a flexible model checking environment for TM systems. ChkTM aims to model the evaluated TM systems at the implementation level to reveal as many potential bugs as possible. Using ChkTM, we found a subtle correctness bug in the current implementation of eager TL2. We also checked the serializability of TL2 and SigTM and strong isolation guarantees of SigTM. We quantitatively analyzed ChkTM to understand the practical issues in model checking TM systems. Finally, our quantitative study motivates further research such as investigating a reduction theorem and dynamic partial order reduction techniques to verify TM systems with support for concurrent nesting without causing a state space explosion.

Chapter 7

Conclusions

This dissertation presents work towards improving the practicality of transactional memory across three dimensions. Specifically, this dissertation makes the following contributions:

- **Improving the programmability of transactional memory**

  We propose *OpenTM*, an application programming interface (API) for transactional programming. We extend a popular parallel-programming environment (OpenMP) to support both non-blocking synchronization and speculative parallelization with TM. OpenTM provides language constructs to integrate memory transactions with OpenMP constructs such as parallel loops and sections. Furthermore, OpenTM defines a set of advanced constructs to address issues such as nesting and contention management. We describe an OpenTM implementation for the C programming language and demonstrate the effectiveness of high-level TM programming with OpenTM.

- **Supporting nested parallelism in transactional memory**

  We propose two TM systems to support nested parallelism within transactions. First, we present *NesTM*, a software TM with support for nested parallel transactions solely in software. Second, we present *filter-accelerated, nested transactional memory* (*FaNTM*) that provides practical support for nested parallel transactions with lightweight hardware. NesTM presents a good model

for concurrent nesting, and FaNTM provides a fast and practical implementation. Specifically, FaNTM extends its baseline hybrid TM system to implement nesting-aware conflict detection and data versioning in a performance- and cost-effective manner. We also quantify the performance of FaNTM across multiple use scenarios using a variety of transactional applications.

- **Implementing a model checker for transactional memory**

  We propose *ChkTM*, a flexible model checking environment for checking the correctness of TM systems. The key design philosophy of ChkTM is to model TM systems close to their implementation level in order to reveal as many potential bugs as possible. In ChkTM, we model several TM systems including a widely-used, high-performance STM system. We describe a case study in which we found a subtle correctness bug in the current STM implementation. We also perform an in-depth, quantitative analysis on ChkTM to understand the practical issues in checking the correctness of TM systems and motivate further research issues.

In summary, this dissertation presents novel techniques and important findings towards improving the practicality of TM across three dimensions. Both TM system designers and application developers can use the techniques and findings described in this dissertation to make TM systems more widely adopted in mainstream parallel programming.

As for future work, it would be interesting to study more transactional applications that can benefit from nested parallelism within transactions. Extensive experiences with a large number of applications would also guide possible enhancements to the current FaNTM design. In addition, it is interesting future work to investigate a nesting-aware runtime system that dynamically exploits the parallelism available in different nesting levels based on runtime information such as the degree of contention. It would be also interesting to utilize the hardware structures of FaNTM for purposes other than support for nested parallelism. For example, multiple hardware signatures per processor core can be effectively used to enable more aggressive compiler and runtime optimizations because the hardware signatures can track memory accesses at

runtime with small performance overheads. The hardware structures that encode the transactional hierarchy in FaNTM can also be used to represent other useful information (e.g., data dependency) to implement more sophisticated contention management schemes.

It is also interesting future work to investigate a reduction theorem and dynamic partial order techniques to reduce the state space when verifying the correctness and other properties of TM systems with support for concurrent nesting. It is also important to incorporate more realistic hardware models such as relaxed consistency and interconnection network into TM model checking environments in order to thoroughly investigate system-level correctness issues.

Bibliography

[1] ARM11 MPCore. http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

[2] The OpenMP Application Program Interface. http://www.openmp.org.

[3] The Scala Programming Language. http://www.scala-lang.org.

[4] Transactional Memory in GCC. http://gcc.gnu.org/wiki/TransactionalMemory.

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

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

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

[8] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium*

on Principles and practice of parallel programming, pages 163–174, New York, NY, USA, 2008. ACM.

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

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

[11] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and Evaluating a Model Checker for Transactional Memory Systems. In *ICECCS '10: Proceedings of the 15th IEEE International Conference on Engineering of Complex Computing Systems*, March 2010.

[12] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *22nd ACM Symposium on Parallelism in Algorithms and Architectures*. June 2010.

[13] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Making nested parallel transactions practical using lightweight hardware support. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 61–71, New York, NY, USA, 2010. ACM.

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

[15] J. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka. Leveraging parallel nesting in transactional memory. In *PPoPP '10: Proceedings*

of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 91–100, New York, NY, USA, 2010. ACM.

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

[17] R. Blikberg and T. Sorevik. Load balancing and OpenMP implementation of nested parallelism. *Parallel Comput.*, 31(10-12):984–998, 2005.

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

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

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

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

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

[23] Broadcom. The Broadcom BCM1250 multiprocessor. In *Presentation at 2002 Embedded Processor Forum*, San Jose, CA, April 2002.

[24] N. G. Bronson, C. Kozyrakis, and K. Olukotun. Feedback-directed barrier optimization in a strongly isolated STM. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–225, New York, NY, USA, 2009. ACM.

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

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

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

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

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

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

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

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

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

[34] A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 37–44. November 2007.

[35] L. Dalessandro and M. L. Scott. Strong isolation is a weak idea. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

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

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

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

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

[40] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM.

[41] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

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

[43] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[44] R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 372–382, New York, NY, USA, 2008. ACM.

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

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

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

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

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

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

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

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

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

[54] G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[55] Intel. First the tick, now the tock: Next generation intel microarchitecture (nehalem). 2008.

[56] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with tla+. *Form. Methods Syst. Des.*, 22(2):125–131, 2003.

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

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

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

[60] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

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

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

[63] B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall, 1998.

[64] C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun. Testing implementations of transactional memory. In *PACT '06: Proceedings*

of the 15th international conference on Parallel architectures and compilation techniques, pages 134–143, New York, NY, USA, 2006. ACM.

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

[66] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, June 2006. IEEE Computer Society.

[67] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–74, Washington, DC, USA, September 2005. IEEE Computer Society.

[68] M. Milovanovic, R. Ferrer, O. Unsal, A. Cristal, X. Martorell, E. Ayguade, J. Labarta, and M. Valero. Transactional memory and openmp. In *International Workshop on OpenMP*, June 2007.

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

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

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

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

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

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

[75] J. O'Leary, B. Saha, and M. R. Tuttle. Model checking transactional memory with spin. *Distributed Computing Systems, International Conference on*, 0:335–342, 2009.

[76] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[77] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 2–11, New York, NY, USA, 1996. ACM.

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

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

[80] H. E. Ramadan and E. Witchel. The Xfork in the Road to Coordinated Sibling Transactions. In *The Fourth ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 09)*, February 2009.

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

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

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

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

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

[86] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing isolation and ordering in

STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

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

[88] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 2006.

[89] C. SPARC International, Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[90] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, Nov. 2001.

[91] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *LCR '00: Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 100–112, London, UK, 2000. Springer-Verlag.

[92] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. Softsig: software-exposed hardware signatures for code analysis and optimization. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 145–156, New York, NY, USA, 2008. ACM.

[93] N. A. Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Princeton University, 2008.

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

[95] G. T. Vassilios Dimakopoulos, Elias Leontiadis. A portable c compiler for openmp v.2.0. In *EWOMP 2003, European Workshop on OpenMP*, 2003.

[96] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *ECOOP*, 2009.

[97] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–89, New York, NY, USA, 2007. ACM Press.

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

[99] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, March 2007.

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

[101] K. C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2), 1996.

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