PROGRAMMING WITH TRANSACTIONAL MEMORY

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

Brian David Carlstrom

June 2008

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) Principal Adviser

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

(Christos Kozyrakis)

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.

(John Mitchell)

Approved for the University Committee on Graduate Studies.

Abstract

For decades the combination of Moore's law and innovations in uni-processor computer architecture have allowed most single-threaded applications to perform better on each subsequent processor generation. However, the power consumption and heat generation of recent generations of complex single-core processors ended this decades long progression and the trend has shifted to multi-core processors, often with simpler cores than found in earlier generations. In order to take advantage of the increasing number of cores found in new processors, single-threaded applications need to be rewritten to be parallel. The most common approach in a multi-core system is to use parallel threads sharing a common address space that use lock-based mutual exclusion to coordinate access to shared data. Unfortunately, writing a correct and scalable program using locks is considerably more difficult than the comparable sequential program. Programming with a few coarse-grained locks often limits scalability while using finer-grained locking often leads to significant overhead and risks issues such as deadlock.

Transactional memory is an alternative to locks for coordinating concurrent access to shared data in parallel programs. By allowing speculative concurrent access to shared data, both software and hardware transactional memory systems have been show to allow more scalability than locks on machines with one hundred or more cores. However, these results have focused on converting the use of short critical sections to transactions in existing parallel applications. Many hope that transactional memory will make parallel programming easier by allowing developers to reason about the interactions between fewer coarse-grained transactions that cover the majority of program execution.

The thesis of this dissertation is that supporting scalable performance similar to fine-grained locks with coarse-grained transactions requires going beyond simple atomic transactions to support transactional conditional waiting, as well as ways of reducing isolation between transactions. The analysis begins with JavaT, a transactional execution model for Java programs. While this model allows for a transactional evaluation of benchmarks such as SPECjbb2000 and JavaGrande, it also shows that there is no single interpretation of monitor-based conditional waiting that preserves the behavior of all existing Java programs. This leads to the definition of the Atomos transactional programming language, a Java derivative that includes features such as transactional conditional waiting, closed- and open-nested transactions, and transaction handlers.

To evaluate Atomos for executing programs with long transactions, a special version of SPECjbb2000 is created that spends most of its execution time executing transactions on a single shared warehouse data structure. The technique of semantic concurrency control is used to create transactional collection classes to enable the scalability of this SPECjbb2000 variant using Atomos nesting and handler support. Several techniques for dealing with non-transactional operations such as I/O are also discussed.

Acknowledgements

Oh, yes, the acknowledgements. I think not.
I did it. I did it all, by myself.
– Olin Shivers [117]

I would like to thank my advisors Kunle Olukotun and Christoforos Kozyrakis. Without them I do not know how I would have translated my vague goal of improving software engineering into something concrete to the challenges facing today's programmers. I'd also like to thank Darlene Hadding for helping me over come the challenges facing today's PhD students at Stanford.

I would like to thank John Mitchell for being on my reading committee and Dawson Engler for being on my orals committee, which brought some continuity to my undergraduate research experience at MIT. I would also like to thank Jef Caers for stepping in at the last minute to be the chair for my oral thesis defense.

I would like to thank Austen McDonald, JaeWoong Chung, Jared Casper, Ben Hertzberg, Hassan Chafi, and Tayo Oguntebi for their work on the simulator that made my experiments possible. Special thanks to JaeWoong Chung, Nathan Bronson, and Mike Carbin for their work on JikesRVM and Atomos. Thanks to the TCC group and Kunle students for all their helpful feedback, advice, and suggestions. Special thanks to Lance Hammond, not only for his work on the old simulator and trace analyzer that got TCC off the ground, but also for the unique advice and perspective that only a senior officemate can provide.

Extra special thanks to Jacob Leverich and Andrew Selle for all those cluster cycles. Thanks to Justy Burdick and Steven Tamm for PowerPC cycles back in the day.

Thanks to my MIT undergraduate research advisors Olin Shivers and Tom Knight for the fun UROP experiences. Thanks to Norman Adams, Marc Brown, and John Ellis for encouraging me to go back to school. Thanks to everyone at Ariba that made going back to school possible.

I would like to thank the Intel Foundation for their support through their PhD Fellowship program. I would also like to thank DARPA, not only for supporting me and my group over the last five years, but my father for my first five years.

Thanks to my parents, David and Elizabeth, for all their support of my education as well as for taking me all over the world to AAAI and IJCAI conferences.

Finally, I would like to thank my wife Jennifer and my children Michael, Daniel, and Bethany for humoring me in my hobby of getting a PhD.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There are many examples of systems that tried and failed to implement fault-tolerant or distributed computations using ad hoc techniques rather than a transaction concept. Subsequently, some of these systems were successfully implemented using transaction techniques. After the fact, the implementers confessed that they simply hit a complexity barrier and could not debug the ad hoc system without a simple unifying concept to deal with exceptions [Borr 1986; Hagmann 1987]. Perhaps even more surprising, the subsequent transaction oriented systems had better performance than the ad hoc incorrect systems, because transaction logging tends to convert many small messages and random disk inputs and outputs (I/Os) into a few larger messages and sequential disk I/Os.

– Jim Gray [47]

Processor vendors have exhausted their ability to improve single-thread performance using techniques such as simply increasing clock frequency [128, 4]. Hence, they are turning *en masse* to single-chip multiprocessors (CMPs) as a realistic path towards scalable performance for server, embedded, desktop, and even notebook platforms [75, 71, 72, 17]. While the trend is to use Moore's Law to continually add more cores to processors, the practical benefits to various applications are unclear.

Several applications workloads clearly benefit from CMPs. Server applications with task-based parallelism have been straightforward to move from older symmetric multiprocessing (SMP) machines. Scientific and engineering workloads with significant data parallelism, as well as multiprogramming workloads, also benefit from the additional computation resources. The key similarity is that there is relatively little coordination needed between parallel threads in these workloads.

On the other hand, adapting desktop application workloads to take advantage of CMPs is more problematic. While some limited multiprogramming such as virus checking exists on the desktop, most applications have historically been single-threaded event based applications. In the past, desktop application performance benefited from simply moving to a new faster processor. If existing desktop application want to improve their performance, they need to find some way to exploit computational parallelism.

While some server workloads can scale by running independent tasks in parallel, increasingly the challenge is to parallelize within a single operation. Where previously global data structures were accessed only by a single thread, now multiple threads will need to coordinate access to this shared data.

1.1 Locks

Multithreaded programming has traditionally used locks to coordinate access to shared data. Lock-based coordination works through the principal of mutual exclusion. A thread that owns a lock can be sure that they are are they only one accessing the associated shared data. The association between locks and the data they protect is by convention, although some languages such as Java try to encourage specific patterns for the common case of using a single lock to protect an object.

Typically, a programmer will use one lock per data structure to keep the locking protocol simple. Unfortunately, such coarse-grained locking often leads to serialization on high-contention data structures. An alternative approach is to use finer-grained locking where more than one lock is associated with a data-structure. For example, a tree may have locks on each node or a hash table may have locks for each bucket. While fine-grained locks can improve the concurrency of parallel operations, they can hurt the latency for a single operation due the additional overhead of selecting locks or acquiring and releasing multiple locks. Fine-grained locking also increases code complexity for operations that inherently touch shared data. For example, maintaining the size property of a tree or hash table in the presence of multiple mutators can significantly complicate the implementation.

Finally, there are many known pitfalls to using locks for coordination such as deadlock, priority inversion, and convoying, which are exacerbated through the use of fine-grained locking. For example, atomically moving a value from one location in a tree or hash table when the whole table is protected by a single lock is trivial in comparison with the deadlock issues that can arise when multiple locks must be held to provide the appearance of atomicity.

1.2 Alternatives to Locks

Because of these issues with locks, alternative forms of *non-blocking synchronization* have been explored. Typically these involve using atomic hardware primitives such as "compare-and-swap" (CAS) directly instead of using them to build locking primitives.

While non-blocking synchronization avoids many issues with locks such as deadlock and priority inversion, it can introduce issues of livelock, starvation, as well as new sources of overhead. Although non-blocking synchronization techniques may be more scalable than locks, in practice correctness is even harder to achieve than with the mutual-exclusion approach of locks because of the many different possible race conditions. Finally, non-blocking techniques often make composing two different operations into one atomic action even harder than with locks.

1.3 Transactional Memory

Transactional memory has been proposed as an abstraction to simplify parallel programming [61, 115, 60, 55]. Transactions eliminate locking completely by grouping sequences of object references into atomic and isolated execution units. They provide an easy-to-use technique for synchronization over multiple operations on multiple objects, allowing the programmer to focus on determining where atomicity is necessary, and not its implementation details.

Much of the earlier work on transactional memory has focused on how replacing lock-based critical sections with transactions can improve concurrency. However, this does not do much to simplify parallel programming since the challenge of selecting the right short atomic sections remains. In this dissertation I will show that transactional memory can truly make parallel programming easier by allowing programmers to focus on executing a few large atomic operations in parallel while maintaining scalability competitive with fine-grained locks. In this style, most of the time is spent running in transactions, which are not limited to short critical sections.

1.4 Thesis

In this dissertation I intend to prove the following thesis:

> Supporting scalable performance similar to fine-grained locks with coarse-grained transactions requires going beyond simple atomic transactions to support transactional conditional waiting, as well as ways of reducing isolation between transactions.

To prove this thesis, I will:

- show the need for transactional conditional waiting because of the correctness problems that come from mixing monitors with transactions
- show the need for alternative transaction types that allow programs to reduce the strict serializability of transactions
- show the need for semantic concurrency control to allow programs to focus on logically meaningful data dependencies between transactions, rather than strictly on memory dependencies
- show mechanisms for integrating non-transactional operations into transactions
- provide qualitative results demonstrating these needs.

1.5 Organization

The remainder of this dissertation is organized as follows:

Chapter 2 discusses the advantages and disadvantages of various programmer interfaces, semantic models, and implementation approaches for transactional memory. This provides the background for the choices made in the transactional memory systems and language used in this dissertation.

Chapter 3 looks at JavaT, a transactional model for the execution of existing Java programs. JavaT reinterprets existing Java language features to have transactional memory semantics. While displaying the performance potential of transactional memory, it also demonstrates the correctness problems when transactional memory is combined with monitor-style conditional waiting.

Chapter 4 describes the design of Atomos, a programming language system built entirely on transactional memory. Here I will show the need for reducing the serializability of transactions in both language implementation and applications.

Chapter 5 shows how semantic concurrency control can improve the concurrency of parallel operations on abstract data types. I show how this is critical to scalability when parallel programs consist of a few large transactions operating on shared data structures.

Chapter 6 concludes the dissertation by revisiting my thesis in the context of the results from the previous chapters. It discusses directions for future work and finishes with some final thoughts on the potential of transactional memory.

Following the dissertation proper, there are three appendices with additional discussion on the implementation of transactional memory systems, focusing primarily on the system used for evaluation in this dissertation. Appendix A covers the details of the simulation environment, including some additional details on the hardware configurations and transactional memory semantics. Appendix B discusses the details of the virtual machine environment I use for running both Java, JavaT, and Atomos experiments. Appendix C describes how non-transactional operations, particularly I/O, can be integrated with transactional memory though a variety of mechanisms such as simple handlers at the end of transactions, general semantic concurrency control, and distributed transactions.

Chapter 2

Choices in the Design of a Transactional Memory System

The secret to creativity is knowing how to hide your sources.
– Albert Einstein

All transactional memory systems need to detect *violations* of data dependencies between transactions. Violations occur when a transaction's *read-set*, the set of all locations read during the transaction, intersects with another transaction's *write-set*, the set of all locations written during the transaction. While this basic idea of violation detection is common to all systems, there are many different choices to be made in the design and implementation of a transactional memory system. In this section I discuss possible choices in the area of implementation, semantics, as well as the choices made in the system used for evaluation in this dissertation.

2.1 Semantics

Transactional memory semantics describe the expected or allowed outcomes of various memory operations which can include both memory transactions and non-transactional loads and stores. Unlike SQL transactions, where there is a clear separation between program data and relation data, transactional memory systems have to cope with potentially unexpected interactions between code that is run inside a transaction and code that is run outside a transaction, as well as interactions between applications and the underlying libraries and runtime system.

2.1.1 Weak Isolation versus Strong Isolation

The isolation criteria defines how transactional code interacts with non-transactional code [80]. In systems with *weak isolation*, transactional isolation is only guaranteed between code running in transactions, which can lead to surprising and non-deterministic results if non-transactional code reads or writes data that is part of a transaction's read- or write-set. For example, non-transactional code may read uncommitted data from the transaction's write-set and non-transactional writes to the transaction's read-set may not cause violations. In systems with *strong isolation*, non-transactional code does not see the uncommitted state of transactions and updates to shared locations by non-transactional code violate transactions, if needed, to prevent data races.

From a programming model point of view, strong isolation makes it easier to reason about the correctness of programs because transactions truly appear atomic with respect to the rest of the program. However, most software implementations of transactional memory have only guaranteed weak isolation as a concession to performance. Recently, some hardware and hybrid systems that support unlimited transaction sizes have also only offered weak isolation. The problem is that programs written for one isolation model are not guaranteed to work on the other; for a transactional program to be truly portable, it has to be written with a specific isolation model in mind, potentially hindering its reuse on other systems [12].

2.2 Interface

Regardless of the semantics of a transactional memory system, there are different types of programming interfaces that can be presented to application programmers ranging from hand annotating each memory access to programming language integration.

2.2.1 Explicit versus Implicit Transactions

Some systems require an *explicit* step to make locations or objects part of a transaction, while other systems make the memory operations' behavior *implicit* on the transactional state. Implicit transactions require either compiler or hardware support [2]. Older systems often required explicit instructions or calls to treat specific locations or objects as transactional; however, most systems now allow existing code to run both transactionally and non-transactionally based on the context. Requiring explicit transactional operations prevents a programmer from composing existing non-transactional code to create transactions. This can create a maintenance burden as programmers need to create and maintain transaction-aware versions of existing non-transactional code in order to reuse it.

2.2.2 Library Interface versus Language Integration

Some systems treat transactions simply as a library, while others integrate transactions into the syntax of the programming language. There are many issues with not properly integrating concurrency primitives with programming language semantics, as shown in recent work on the Java Memory Model and threads in C and C++ [104, 15]. Clear semantics are necessary to allow modern optimizing compilers to generate safe yet efficient code for multiprocessor systems as well as perform transactional memory specific optimizations [57, 2].

2.3 Implementation

Perhaps the greatest diversity in transactional memory systems is in the variety of implementation approaches found in competing systems. As with database systems, there are a variety of ways to provide the transactional properties of atomicity and isolation.

2.3.1 Eager versus Lazy Versioning

All transactional systems must be able to deal with multiple versions of the same logical data. At the very least, systems need to be able to deal with two versions: a new updated version and an old version to use in case the transaction fails to commit. In systems with *eager versioning*, the new version is stored "in place", while the old version is stored in an undo log. Since the new version is stored in place of the old version, there can only be one new version at a time. In systems with *lazy versioning*, the old version remains in place and new versions are placed in a per transaction store buffer or redo log. A new version replaces the old version when a transaction commits.

Systems with eager versions require *contention management* to prevent deadlock. If two transactions are contending to write to the same location, only one can succeed. If the first transaction to reach the location always is granted exclusive access, it is possible that transaction X may write location A and transaction Y may write location B. Then if transaction X needs to write location B and transaction Y needs to write location A, a classic deadlock situation is created. To avoid this, a contention manager can decide to detect a potential deadlock cycle and break the deadlock by choosing a victim to rollback.

Systems with lazy versioning also require contention management, in this case to avoid livelock. If the system detects conflicts between transactions at commit time and always lets transactions commit in "first come, first served" order, then short transactions may repeatedly violate a longer running transaction, preventing it from ever completing. A contention manager could detect the repeated violation and allow the older transaction to complete to avoid *starvation.*

Much of the work on contention management studies the usefulness of various contention management policies to avoid certain performance pathologies on various styles of workloads. Like database systems, there are often common case performance trade-offs to be made. For example, the graph algorithms necessary for deadlock prevention are often too expensive to run for each written location when transactions have few conflicts. In the case of databases, a simple timeout mechanism is used to detect transactions that have been waiting for more than some threshold, at which point deadlock detection can be run and deadlock recovery can be initiated by rolling back a transaction involved in the deadlock cycle.

2.3.2 Optimistic versus Pessimistic Conflict Detection

As alluded to in the discussion of versioning, when data dependencies are detected can vary based on versioning implementation. Systems with *pessimistic conflict detection* notice possible data dependency violations as soon as possible. In the example of eager versioning, this is necessary to make sure that only one transaction has exclusive access to write a new version. Systems with *optimistic conflict detection* do not worry about detecting conflicts as early as possible, but wait until a transaction commits to detect conflicts. As suggested, this is a common approach in systems with lazy versioning since they do not need to detect conflicts until transactions start committing. However, these are not the only two alternatives. Some systems provide lazy versioning with pessimistic conflict detection. On the other hand, eager versioning with optimistic conflict detection is illogical because without independent storage for multiple versions, conflicts need to be detected as soon as possible so that only one transaction will proceed.

2.3.3 Uniprocessor versus Multiprocessor

Some systems require a uniprocessor implementation for correctness, while others take advantage of multiprocessor scaling. Since trends indicate a move to multiprocessors, new programming languages should make it easy to exploit these resources. To properly evaluate transactional memory as an abstraction to simplify parallel programming, it is important for systems to provide a multiprocessor implementation.

2.3.4 Hardware versus Software

Finally, one of the biggest choices in implementing a transactional memory system is the use of hardware support. The earliest transactional memory proposals came out of the computer architecture community and would now be classified as hardware transactional memory (HTM) systems [74, 61]. However, it was not long before the first software transactional memory (STM) system was proposed [115]. Recently there has been a resurgence in both HTM and STM systems. For hardware, this has largely been fueled by the growing importance of making chip multiprocessors programmable and for software, this has been inspired by a number of new STM algorithms coming from the non-blocking synchronization community [55, 40].

The primary advantage of hardware transactional memory is speed. A software transactional memory adds significant overhead to memory operations. Although compiler optimization can eliminate some of this overhead, especially in the case of weak isolation. However, if strong isolation is desired even memory references outside of transactions can have overhead. Since the hardware transactional memory systems are typically based on cache coherence protocols, they do not suffer additional overhead in supporting strong isolation.

Another important advantage of HTM systems is that legacy code, such as C libraries, can be called from transactional code without loss of atomicity since the hardware detects data dependency violations regardless of source language. Since STM systems require programmer or compiler annotations, transactional memory is only provided to code managed by or annotated for the STM system but not legacy code.

The primary advantage of software transactional memory is semantic flexibility. It is easy to change implementations to pick the semantics right for a particular application. For example, one particular workload might perform better with a lazy optimistic system while another might prefer an eager pessimistic system. Another example is that some applications may want strong isolation and others are fine with weak isolation. Additionally, a specific application might want a TM implementation with extended semantic features. For example, one application might need to use composable conditional waiting while another might simply need simple nested transactions. With an HTM, typically certain choices such as conflict detection, versioning, and isolation guarantees, are baked into the hardware implementation. Other things such as support for transactional conditional waiting might be able to be implemented on top of a lower level interface if it provides the right primitives.

Another advantage of software transactional memory is resource flexibility. Typically HTM systems have fixed resources that limit the size and number of concurrent transactions, whereas STM systems are limited only by what memory is available in the system. While a number of proposals exist to virtualize HTM resources, the performance benefits of HTM are dependent on a given application's ability to fit within the resource limits of the hardware in the common case. It is worth noting that additional cores and larger caches fueled by the new transistors provided by Moore's Law are likely to lessen this advantage of STM systems over time.

Another benefit of STM systems is granularity detection. Many STM systems can detect data dependencies at the granularity of individual words. HTM systems that depend on cache coherence protocols typically only detect data dependencies at cache line granularity, which typically is eight words on today's systems. Detecting dependencies at this coarser cache line granularity means that two transactions updating adjacent but unrelated fields in the same object will be detected as a violation. Similar to the problem of false sharing, these *false violations* hurt performance but not correctness.

Because of the advantage and disadvantages of each type of system, there are hybrid approaches that try to combine the benefits of both worlds [19, 78, 35, 111]. While most hybrid systems try to focus on accelerating the common case, some try to provide additional benefits such as strong isolation. Although not typically considered hybrid systems, most HTM systems have now moved to implement functionality such as contention management in software to allow more flexible policy decisions.

2.4 Evaluation Environment

Out of the many possible choices in creating a transactional memory system, the one used as the baseline configuration for this dissertation has the following properties:

- strong isolation
- implicit transactions
- programming language integration

- lazy versioning
- optimistic conflict detection
- multiprocessor implementation
- hardware transactional memory

The choices of strong isolation, implicit transactions, and programming language integration were made based on what is important for the novice programmer converting sequential code to parallel code for the first time. Strong isolation was chosen to help prevent surprises when there are non-transactional accesses to shared data. Implicit transactions make it easy to reuse existing code and libraries, allowing the programmer to focus on their parallel application and not creating new transactional libraries and runtime. Programming language integration eliminates unexpected interactions with compiler optimizations.

The choices of lazy versioning, optimistic conflict detection, multiprocessor implementation, and hardware transactional memory were made based on what would provide the best performance for a HTM system [14]. If a programmer does not see benefits from transactional memory in this environment, they are unlikely to see benefits in a software-based system.

Although STM systems may provide more flexibility, the overhead during transactions makes them more appropriate for replacing short critical sections for improved scalability. Since my goal is to encourage programmers to use transactions for most of the parallel program execution, hardware transactional memory makes sense from a software engineering point of view as well.

The software environment used to provide programming language integration is the Jikes Research Virtual Machine (JikesRVM) version 2.3.4. JikesRVM, formerly known as the Jalapeño Virtual Machine, has performance competitive with commercial virtual machines and is open source [7]. It provides an opportunity to explore not just how transactions can benefit applications, but how they can impact the runtime environment as well. For more details on the virtual machine environment see Appendix B.

The HTM chosen to execute JikesRVM is a lazy, optimistic system described in [90]. This system is an x86 simulator that provides a chip-multiprocessor configuration with up to 32 processors with speculative private L1 and L2 caches as well as a shared non-speculative L3 cache. For comparison with lock-based applications, the simulator also provides a MESI cache coherence environment with the same cache hierarchy and parameters. For more details on the simulation environment see Appendix A.

2.5 Related Work

Recent transactional hardware systems build on earlier hardware transactional memory work as well as thread-level speculation (TLS). Java programming with transactions builds on earlier work on speculating through locks and transactional memory in general.

2.5.1 Early Hardware Transactional Memory

Hardware transactional memory variants have now been around for almost twenty years and have focused on multiprocessor implementations. Knight first proposed using hardware to detect data races in parallel execution of implicit transactions found in mostly functional programming languages such as Lisp [74]. This proposal had two of the important features of transactional memory: implicit transactions and strong isolation. However, the transaction granularity was not made visible to the programmer, with each store acting as a commit point and executing in sequential program order. Herlihy and Moss proposed transactional memory as a generalized version of load-linked and store-conditional, meant for replacing short critical sections [61].

2.5.2 Thread Level Speculation

Thread level speculation (TLS) uses hardware support to allow speculative parallelization of sequential code [52, 76, 120, 123]. TLS systems are similar to HTM systems in that they provide versioning and prevent data dependency violations. TLS systems allow only ordered execution, as opposed to HTM systems that typically allow threads to commit in any order. Because TLS systems maintain a sequential execution order, there is always one thread that is non-speculative. TLS systems allow speculative threads to communicate results to other speculative threads continuously. This communication, known as *forwarding*, allows more speculative threads to see results from less speculative threads. By comparison, HTM systems threads only see committed data.

2.5.3 Speculating through Locks

Rajwar and Goodman proposed Speculative Lock Elision (SLE) [105]. SLE speculates through lock acquisition, allowing concurrent execution using hardware support. If a data dependency is detected, all involved processors roll back. Later, Rajwar and Goodman extended SLE to create Transactional Lock Removal (TLR) which used timestamps to avoid rolling back all processors, giving priority to the oldest outstanding work [106].

Martínez and Torrellas proposed Speculative Synchronization [87] based on TLS hardware. It supports speculating not just through locks, but also barriers and flags. These systems have the concept of a *safe thread* that is non-speculative, and that thread has priority when data dependencies are detected, similar to TLR's use of the oldest timestamp.

SLE, TLR, and Speculative Synchronization build upon conventional cache coherence and consistency, similar to early hardware transactional memory and TLS. TLR and Speculative Synchronization do not provide a way for the programmer to override the forward progress guarantee to improve performance by using completely unordered transactions in cases of load imbalance between threads.

The idea of speculating through locks has found its way into commercial hardware such as systems with Vega processors from Azul Systems [31], as well as the upcoming Rock processor from Sun Microsystems [125, 126].

2.5.4 Recent Hardware Transactional Memory

Recent systems such as TCC, UTM/LTM, VTM, XTM, and LogTM have relieved earlier data size restrictions on transactions, allowing the development of continuous transactional models [53, 8, 107, 29, 92]. TCC provides implicit transactions, strong isolation, and some features for speculation and transactional ordering [51]. UTM and LTM are related systems that both provide implicit transactions while focusing on transactions larger than the available cache size. However, UTM and LTM differ in isolation guarantees, with UTM providing weak isolation and LTM providing strong isolation [11]. VTM provides implicit transactions and strong isolation, again with a focus on supporting larger transactions. LogTM provides an eager versioning system with pessimistic conflict detection, in contrast to the other systems mentioned which have lazy versioning and optimistic conflict detection.

Most HTM systems generally layer speculative execution on top of a conventional cache coherence and consistency protocol. In contrast, TCC completely replaces the underlying coherence and consistency protocol, and transactions only make their write state visible *only* at commit time, attempting to replace many small latency sensitive coherence operations with fewer bandwidth sensitive operations, hopefully taking advantage of the scaling of chip multiprocessor systems.

From a programming model perspective, most transactional memory systems do not allow program control over transaction order. TCC allows unordered and ordered transactions, including the use of both models simultaneously.

The preemption of processors with transactional state is a general problem faced by hardware transactional memory systems. VTM addresses this issue by allowing all of the hardware state associated with a transaction to be stored in virtual memory, treating a context switch similar to an overflow of the hardware state. Software transactional memory systems do not face any problems with preemption because their transactional state is already stored in virtual memory.

2.5.5 Software Transactional Memory

Shavit and Touitou first proposed a software-only approach to transactional memory, but it was limited to static transactions where the data set is known in advance, such as k-word compare-and-swap [115]. Herlihy et al. overcame this static limitation with their dynamic software transactional memory work, which offered a Java interface through library calls [60]. Harris and Fraser provide language support for software transactional memory, allowing existing Java code to run as part of a transaction and providing an efficient implementation of Hoare's conditional critical regions (CCRs) [55].

Unlike Shavit and Touitou, later systems support dynamic transactional memory. Unlike Herlihy et al., Harris and Fraser can run *unmodified* code within transactions. Unlike Harris and Fraser, hardware transactions can run both Java and native code within a transaction, as well as support non-transactional operations within atomic regions.

2.5.6 Semantics

Transactional Featherweight Java explores the soundness and serializability properties of transactional memory models with both optimistic and pessimistic conflict detection [127, 66]. Harris et al. provide an operational semantics for the Haskell STM system as part of introducing support for their `orElse` and `retry` constructs. Wojciechowski provides an operational semantics for an isolation-only transaction system that avoids the need for runtime rollback through compile-time checking [132]. Liblit provides a much lower level assembly language operation semantics for the LogTM system [83]. Moore and Grossman provide operations semantics for a variety of possible transactional memory systems exploring strong isolation and several versions of weak isolation as well as three different thread creation semantics including nested parallelism [93]. Abadi et al. provide the semantics of a new transactional memory programming model called Automatic Mutual Exclusion, which creates parallelism through asynchronous methods calls, rather than traditional threads [1].

Chapter 3

JavaT: Executing Java with TM

He who every morning plans the transaction of the day and follows out that plan, carries a thread that will guide him through the maze of the most busy life. But where no plan is laid, where the disposal of time is surrendered merely to the chance of incidence, chaos will soon reign.
– Victor Hugo

Java is a modern, object-oriented programming language used for a range of application workloads ranging from embedded devices to desktops to servers. One notable feature of Java is that, unlike earlier languages such as C or C++, it was built from the start with support for parallel programming. Combined with the fact that Java is widely taught to undergraduate students, Java seems ideal as a starting place for a transactional program language for novice parallel programmers.

This chapter discusses JavaT, a mapping of Java concurrency features to a transactional execution model in order to run existing Java programs with transactions. The discussion includes how transactions interact with the Java memory model, the Java Native Interface, non-transactional operations, and exceptions. The impact of transactions on Java programs is evaluated as well as the impact on virtual machines used to run these programs.

```
public static void main(String[] args){
    a();                    // a(); non-transactional
    synchronized(x){        // BeginNestedTX();
        b();                // b();     transactional
    }                       // EndNestedTX();
    c();                    // c(); non-transactional
}                           //
```

Figure 3.1: Converting a `synchronized` statement into transactions.

3.1 JavaT: Mapping Java to Transactions

This section discusses JavaT's mapping of various existing Java constructs into transactions.

3.1.1 synchronized Statements

Java provides lock-based mutual exclusion though the `synchronized` statement which consists of a lock expression and a block of statements to execute under the protection of a lock. To execute a `synchronized` statement, the lock expression is evaluated to yield an `java.lang.Object`, the lock associated with that object is acquired while the block of statements is executed, after which the lock is released. When `synchronized` statements are nested, each block protects its associated statements with the inner lock protecting the inner statements and the outer lock held over all of its statements, including the inner `synchronized` statement. Some notable differences from other lock implementations are that every object has an associated lock so there is no need for separate lock type as well as the block structure ensures that lock releases are always pared with lock acquires.

Creating transactions from synchronized statements

When running with the JavaT model, `synchronized` statements are used to mark transactions within a thread. A `synchronized` statement defines three transaction regions: the non-transactional region before the statement, the transaction within

```
public static void main(String[] args){
    a();                    // a(); non-transactional
    synchronized(x){        // BeginNestedTX();
        b1();               // b1(); transaction at level 1
        synchronized(y){    // BeginNestedTX();
            b2();           // b2(); transaction at level 2
        }                   // EndNestedTX();
        b3();               // b3(); transaction at level 1
    }                       // EndNestedTX();
    c();                    // c(); non-transactional
}                           //
```

Figure 3.2: Converting nested `synchronized` statements into transactions.

the block, and the non-transactional region after the statement. As an example of how transactions are created in a Java program, consider the simple program shown in Figure 3.1. This program creates a single transaction surrounded by regions of non-transactional code execution.

When `synchronized` statements are nested in JavaT, either within a method or across method calls, the results of execution are only visible to other threads when the outermost `synchronized` statement completes. This is referred to as a closed nesting model [96]. As an example, consider the simple program shown in Figure 3.2. The results of the child transaction nested at level 2 are only visible to other threads when the parent transaction at level 1 completes.

Handling these nested transactions is important for composability. It allows the atomicity needs of a caller and callee to be handled correctly without either method being aware of the other's implementation. The block structured style of `synchronized` is fundamental to this, as it ensures that transaction begins and ends are properly balanced and nested. Simply exposing a commit method to programmers would allow a library routine to commit arbitrarily, potentially breaking the atomicity requirements of a caller.

While nested transactions are necessary for composability, the runtime flattening into a single transaction is not the only possibility. The database community has explored alternatives, including nested transactions allowing partial rollback. Both

HTM and STM systems can provide partial rollback [88, 101]. In fact, the need for partial rollback support in building semantic concurrency control will be shown in Chapter 5.

Lock expression unused in JavaT

Although `synchronized` statements are used to mark transactions, the actual lock object specified is not used. That is because the transactional memory system will detect *any* true data dependencies at runtime. In a purely transactional variant of Java, one can imagine replacing `synchronized` statements with a simpler `atomic` syntax omitting the lock variable as is done in other systems [55, 6, 27, 24]. In such a system, programmers would not need to create a mapping between shared data and the lock objects that protect them. Chapter 4 will look at the Atomos programming language and compare its `atomic` syntax with Java `synchronized` statements.

The fact that the lock variables are not used points to a key advantage of transactions over locks. In Java without transactions, there is not a direct link between a lock object and the data it protects. Even well intentioned programs can mistakenly synchronize on the wrong object when accessing data. With transactions, *all* data accesses are protected, guaranteeing atomic semantics in all cases. There is no reason for basic data structures to provide any synchronization, because the caller defines its own atomicity requirements. Hence, programmers can write data structures without resorting to complex fine-grained locking schemes to minimize the length of critical sections.

Benefits of JavaT execution

The evolution of Java collection classes show how locking can complicate one of the most commonly used data structure classes: the hash table. Java's original `Hashtable` used `synchronized` to guarantee internal consistency, which is important in a sandbox environment. However, in JDK 1.2, a simpler non-locking `HashMap` was introduced, since most applications needed to avoid the overhead of the implicit locking of the original `Hashtable`. Recently, JDK 1.5 has complicated matters further by adding a

```
String intern () {
  synchronized (map){
    Object o=map.get(this);
    if (o!=null){
      return (String)o;
    }
    map.put(this,this);
    return this;
  }
}
```

Figure 3.3: A string interning example.

```
void handleRequest(
  String id,
  String command){
    synchronized (sessions){
      Session s =
        sessions.get(id);
      s.handle(command);
      sessions.put(id,s);
    }
}
```

Figure 3.4: Synchronizing on a `Session`.

`ConcurrentHashMap` that allows multiple concurrent readers and writers.

A transactional memory model eliminates the need for this kind of complexity in the common case. Consider the simple string interning example in Figure 3.3. With JavaT transactional execution, there is no need to use anything other than the non-locking `HashMap` since the caller specifies its atomicity requirements, creating a single logical operation out of the `Map` `get` and `put` operations. Concurrent reads to the `map` can happen in parallel. In addition, non-conflicting concurrent writes can happen in parallel. Only conflicting and concurrent reads and writes cause serialization and this is handled automatically by the transactional memory system, not the programmer.

Traditionally, users of `synchronized` statements are encouraged to make them as short as possible to minimize blocking other threads' access to critical sections. The consequences of making the critical section too large is that processors often spend more time waiting and less time on useful work. At worst, it can lead to complete serialization of work. Consider the example code in Figure 3.4. Because of the way this code is written with a `synchronized` statement around the entire routine, a multithreaded web server becomes effectively single threaded, with all requests pessimistically blocked on the `sessions` lock even though the reads and writes are non-conflicting. Because a transactional system such as JavaT can optimistically speculate through the lock, it does not have this serialization problem. Non-minimally sized transactions do not cause performance problems unless their is actual contention

```
synchronized int get(){
  while (available == false)
    wait();
  available = false;
  notifyAll();
  return contents;
}
```

Figure 3.5: `get` code used by the consumer.

```
synchronized void put(int i){
  while (available == true)
    wait();
  contents = i;
  available = true;
  notifyAll();
}
```

Figure 3.6: `put` code used by the producer.

within the transactions, unlike the serialization problems caused by a mutual exclusion approach based on locks.

3.1.2 Object wait, notify, notifyAll Methods

When Java threads need exclusive access to a resource, they use `synchronized` statements. When Java threads need to coordinate their work, they use `wait`, `notify`, and `notifyAll`. Typically, these condition variable methods are used for implementing producer-consumer patterns or barriers.

Consider the example of a simple producer-consumer usage of `wait` and `notifyAll` derived from [18] shown in Figure 3.5 and in Figure 3.6. This code works in Java as follows. When a consumer tries to `get` the `contents`, it takes the lock on the container, checks for `contents` to be `available`, and calls `wait` if there is none, releasing the lock. After returning from `wait`, the caller has reacquired the lock but has to again check for `contents` to be `available` since another consumer may have taken it. Once the data is marked as taken, the consumer uses `notifyAll` to alert any blocking producers that there is now space to `put` a value. An analogous process happens for producers with `put`.

Transactional semantics of Object.wait

In JavaT, the `get` method is `synchronized` so the method is run as a transaction. If there is no data `available` and the reader needs to `wait`, it commits the transaction as if the `synchronized` statement was closed. This is analogous to the semantics of `wait` releasing the lock and making updates visible. When the thread returns from waiting, it starts a new transaction.

Because JavaT commits on `wait`, it also commits state from any outer `synchronized` statements, potentially breaking atomicity of nested locks. One alternative considered was using rollback, since that would preserve the atomicity of outer `synchronized` statements and works for most producer-consumer examples. However, many commonly used patterns for barriers would not work with rollback. Rollback prevents all communication, but the existing Java semantics of `wait` are to release a lock and make any changes visible. This loss of atomicity in outer `synchronized` statements because of nesting is a common source of problems in existing Java programs as well [113].

Fortunately, the nesting of `wait` in `synchronized` statements is rare, since it causes problems in existing Java programs as well. Java programmers are advised not to place calls to `wait` within nested `synchronized` statements because when a thread waits, it retains all but one lock while it is asleep [113]. By their very nature, condition variables are used to coordinate the high-level interactions of threads, so it is rare for them to be used deeply in library routines. For example, a survey of the Apache Jakarta Tomcat web server version 5.5.8 does not reveal any `wait` calls nested within an outer `synchronized` statement. Tomcat does have libraries for purposes such as producer-consumer queuing that include uses of `wait` on an object with a corresponding `synchronized` statement on the same object, but they are used only for high-level dispatching and not called from `synchronized` statements on other objects. A further example is in SPECjbb2000, which has one example of a barrier where committing works and rollback would fail by causing all threads to `wait`.

The handling of `wait` is the thorniest problem in the transactional execution of Java programs. If the system treats `wait` as a rollback, it then provides composable transactional semantics but existing programs will not run. If the system treats `wait` as a commit, it is easy to come up with contrived programs that will not match the previous semantics. However, surveys of benchmarks and open-source systems have not found an example that exhibits a problem treating `wait` as commit. In a programming language built with transactions in mind, the rollback semantics would make more sense [27, 24]; all of the problematic examples requiring commit could be rewritten to use rollback semantics and the atomicity of parent transactions would be preserved.

3.1.3 volatile Fields

Java allows a field to be marked `volatile` which ensures that writes to the field are visible to other threads and that reads see the latest value as well as providing some ordering guarantees relative to other reads and writes. In JavaT, accesses to volatile fields are treated as small transactions to ensure that top level updates are visible to other threads. Unfortunately, when a volatile field is updated within a transaction created by a `synchronized` statement, the update is deferred until the end of the parent transaction. Fortunately, usually volatile fields are used as an alternative to `synchronized` statements. Section 4.1.3 will describe how open-nested transactions provide an alternative when immediate communication of volatile field updates is required regardless of context.

3.2 Impact of Transactions on Java

This section discusses how transactions relate to several aspects of the Java programming language.

3.2.1 Java Memory Model

A new Java memory model was recently adopted to better support various shared memory consistency models [3, 104, 68]. The new model has been summarized in [55] as follows:

> if a location is shared between threads, either:
>
> (i). all accesses to it must be controlled by a given mutex, or
>
> (ii). it must be marked as volatile.

These rules, while overly simplified, are an excellent match for transactional execution as well as an easy model for programmers to understand. The Java memory model ties communication to `synchronized` and `volatile` code. In JavaT, these same constructs are used to create transactions working with shared data.

Run-time violation detection can be used to detect programs that violate these rules, an advantage of systems with strong isolation. For example, non-transactional code might be violated, indicating that a location is shared between threads without being protected by a `synchronized` statement or `volatile` keyword. Alternatively, a transaction can be violated by non-transactional code. While the strong isolation property guarantees that such violations are detected, other systems that only offer weak isolation do not define the interaction between code inside transactions and code outside transactions. Therefore, in the interest of portability, a system with strong isolation may want to report violations between transactional and non-transactional code.

3.2.2 Java Native Interface

The Java Native Interface (JNI) allows Java programs to call methods written in other programming languages, such as C [82]. Software transactional memory systems typically forbid most native methods within transactions [55]. This is because those systems only can control the code compiled by their own compiler and not any code within pre-compiled native libraries. While some specific runtime functions can be marked as safe or rewritten to be safe, this places a lot of runtime functionality in the non-transactional category.

This limitation of software transactional memory systems destroys the composability and portability of Java software components. Code within `synchronized` statements cannot call methods without understanding their implementation. For example, an application using JDBC to access a database needs to know if the driver uses JNI. Even “100% Pure Java” drivers may use `java.*` class libraries that are supported by native code. Therefore, code carefully written to work on one Java virtual machine may not work on another system, since the language and library specifications only consider nativeness to be an implementation detail and not part of the signature of a method.

On the other hand, hardware transactional memory systems do not suffer from this limitation, since they are source-language neutral. Transactions cover memory accesses both by Java *and* native code. This allows programmers to treat code that traverses both Java and native code together as a logical unit, without implementation-specific restrictions.

3.2.3 Non-transactional Operations

Certain operations are inherently non-transactional, such as I/O operations. Many transactional memory systems simply consider it an error to perform a non-transactional operation within a transaction. This certainly would prevent correct execution of existing Java programs.

A simple approach is to allow only one indivisible transactions to run non-transactional operations at a time. Other transactions that do not need to run non-transactional operations may continue to execute in parallel, but cannot commit. Once the indivisible transactions that performed the non-transactional operation commit, other transactions may then commit or begin their own non-transactional operations. This approach generally works with current Java programs. In contrast, the other, more advanced approaches presented below require changes in applications, libraries, or the underlying runtime. JavaT takes the simple approach of serializing non-transactional operations with more sophisticated approaches to be discussed in Appendix C.

There are several classes of non-transactional operations. Most non-transactional operations are concerned with I/O. However, some operations, such as asking for the current time, may be deemed safe for use within transactions without being considered "non-transactional" operations. Another common non-transactional operation is thread creation. Although alternative and more complex models could support rolling back of thread creation, this is of little practical value. As in the above example of nesting conditional waiting within existing transactions, it seems better to consider

```
try {
    a();                a();              a();
    synchronized(x){    BeginNestedTX();  BeginNestedTX();
        b1();           b1();             b1();
        b2();           b2();             b2();
        b3();           b3();
    }                   EndNestedTX();    EndNestedTX();
    c();                c();
}
catch (IOException e){
    d();                                  d();
    synchronized(y){                      BeginNestedTX();
        e1();                             e1();
        e2();                             e2();
        e3();
    }                                     EndNestedTX();
    f();
}
finally {
    g();                g();              g();
}
```

a.) Example Code b.) No Exceptions c.) Exceptions

Figure 3.7: Example of intermixing transactions and exceptions. a.) A `try-catch` construct containing `synchronized` statements. b.) Runtime transactions created by the non-exceptional case from code. c.) Runtime transactions created when exceptions are thrown from `b2()` and `e2()`.

thread creation a high-level operation that rarely occurs within nested `synchronized` statements and forbid its use to simplify the JavaT model.

3.2.4 Exceptions

Transactions and exceptions as treated as orthogonal mechanisms. Most exceptions in practice are `IOExceptions` or `RuntimeExceptions`. Since the rollback of non-transactional operations cannot be guaranteed, the exception handling follows the flow of control as expected by today's Java programmers. Figure 3.7a illustrates an

Feature	**Transactional interpretation**
`synchronized` & `volatile`	nested transactions
`Object.wait`	transaction commit in addition to `wait`
`Object.notify[all]`	no change
Java Memory Model	simplified interpretation from [55]
Java Native Interface (JNI)	no change
I/O operations	serialize I/O transactions
`Thread.start`	only allowed outside of `synchronized`
Exceptions	no change

Table 3.1: Rules for the execution of Java programs with transactional memory.

example. If no exceptions are thrown, the code produces the transactions shown in Figure 3.7b. On the other hand, if `b2()` throws an `IOException` and `e2()` throws a `RuntimeException`, the code is equivalent to Figure 3.7c. This is exactly the same control flow as current Java exception handling.

3.3 Evaluation of JavaT

In this section the performance of existing Java programs running on a traditional multiprocessor using snoopy cache coherence is compared with the same applications converted to run on a chip multiprocessor supporting a lazy versioning, optimistic conflict detecting hardware transactional memory. Table 3.1 provides a summary of the JavaT rules described in Section 3.1 and Section 3.2. Details about the CMP simulator and the JikesRVM environment can be found in Appendix A and Appendix B respectively.

3.3.1 Benchmarks

The collection of benchmarks summarized in Table 3.2 was used to evaluate running applications with both Java and JavaT semantics. The single-processor version with locks is used as the baseline for calculating the percentage of normalized execution time, with a lower percentage indicating better performance.

The micro-benchmarks are based on transactional memory work from Hammond [51]

Benchmark		**Description**	**Input**	**Lines**
TestHistogram	[51]	histogram of test scores	8,000 scores	331
TestHashtable	[55]	threaded Map read/write	4,000 `get`, 4,000 `put`	398
TestCompound	[55]	threaded Map swaps	8,000 swaps	445
SPECjbb2000	[121]	Java Business Benchmark	736 transactions	30,754
Series	[67]	Fourier series	100 coefficients	413
LUFact	[67]	LU factorization	320×320 matrix	1,074
Crypt	[67]	IDEA encryption	32,000 bytes	690
SOR	[67]	successive over relaxation	100×100 grid	327
SparseMatmult	[67]	sparse matrix multiplication	5000×5000 matrix, 100 iter	321
MolDyn	[67]	N-body molecular dynamics	256 particles, 10 iter	930
MonteCarlo	[67]	financial simulation	160 runs	3,207
RayTracer	[67]	3D ray tracer	32x32 image	1,489

Table 3.2: Summary of benchmark applications including source, description, input, and lines of code.

and Harris [55]. SPECjbb2000 [121] was included as a server benchmark while Java Grande kernels and applications provide numerical benchmarks [67].

3.3.2 TestHistogram

TestHistogram is a micro-benchmark to demonstrate transactional programming from Hammond [51]. Random numbers between 0 and 100 are counted in bins. When running with locks, each bin has a separate lock to prevent concurrent updates. When running with JavaT, each update to a bin is one transaction.

Figure 3.8 shows the results from TestHistogram. While the locking version does exhibit scalability over the single-processor baseline, the minimal amount of computation results in significant overhead for acquiring and releasing locks, which dominates time spent in the application. The transactional version eliminates the overhead of locks and demonstrates scaling to 8 CPUs; transactions allow optimistic speculation while locks caused pessimistic waiting. However, at 16 CPUs lock scalability starts to be limited by the communication limits of a shared bus due to the high communication-to-computation ratio. Transaction performance also degrades at 8 CPUs, again due to the high communication-to-computation ratio, which is compounded by the fact that the evaluation HTM does not have a parallel commit protocol, causing the increasing number of threads to simply serialize during their commit

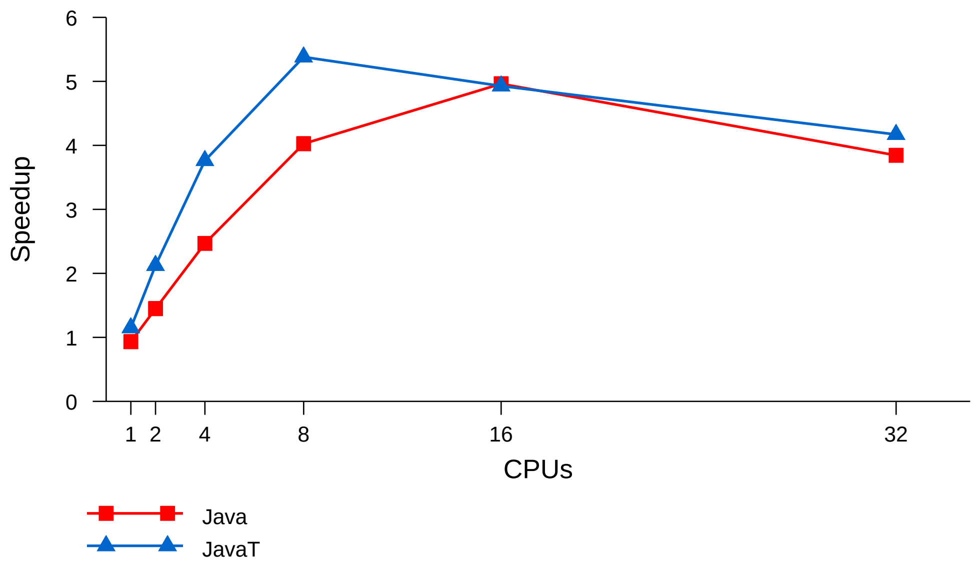


Figure 3.8: TestHistogram creates a histogram of student test scores. Results compare speedup of locks and transactions between 1–32 CPUs.

phase. Parallel commit protocols to scale HTM implementations have been explored in other work [26].

3.3.3 TestHashtable

TestHashtable is a micro-benchmark that compares different `java.util.Map` implementations. Multiple threads contend for access to a single `Map` instance. The threads run a mix of 50% `get` and 50% `put` operations. The number of processors is varied from 1 to 32 and speedup is measured over the single-processor case with locks.

When running with Java locks, three different Map implementations are compared: the original synchronized `Hashtable`, a `HashMap` synchronized using the `Collections` class's `synchronizedMap` method, and a `ConcurrentHashMap` from `util.concurrent` Release 1.3.4 [81]. `Hashtable` and `HashMap` use a single mutex, while `ConcurrentHashMap` uses finer grained locking to support concurrent access. When running with JavaT transactions, each `HashMap` operation is within one transaction.

Figure 3.9 shows the results from TestHashtable. The Java results using locks for

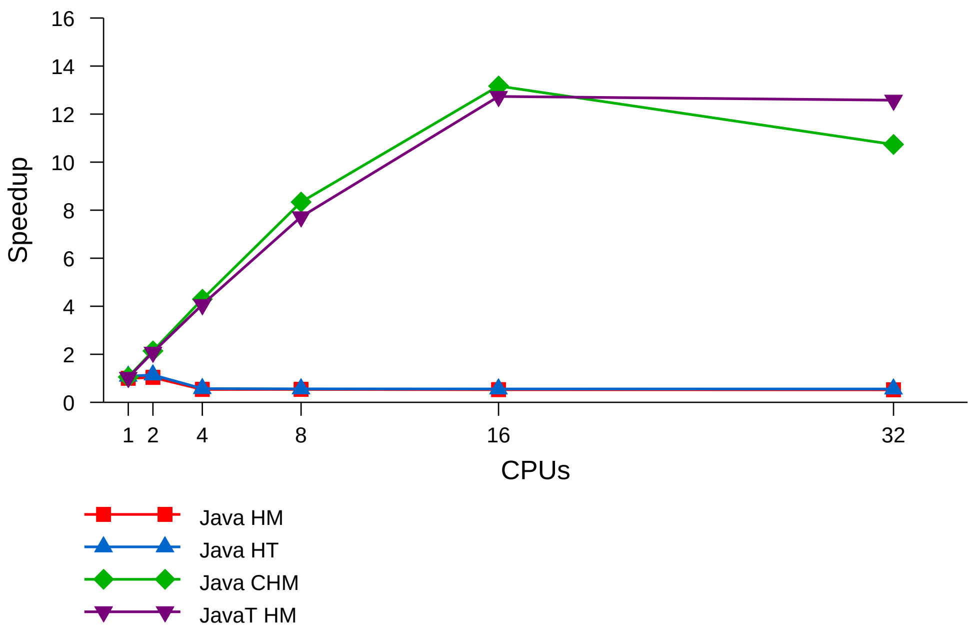


Figure 3.9: TestHashtable performs a 50%-50% mix of `Map` `get` and `put` operations. Results compare various `Map` implementations between 1–32 CPUs. Key: Java HM=`HashMap` with a single caller lock, Java HT=`Hashtable` with a single internal lock, Java CHM=`ConcurrentHashMap` with fine-grained internal locks, JavaT HM=`HashMap` with a single transaction.

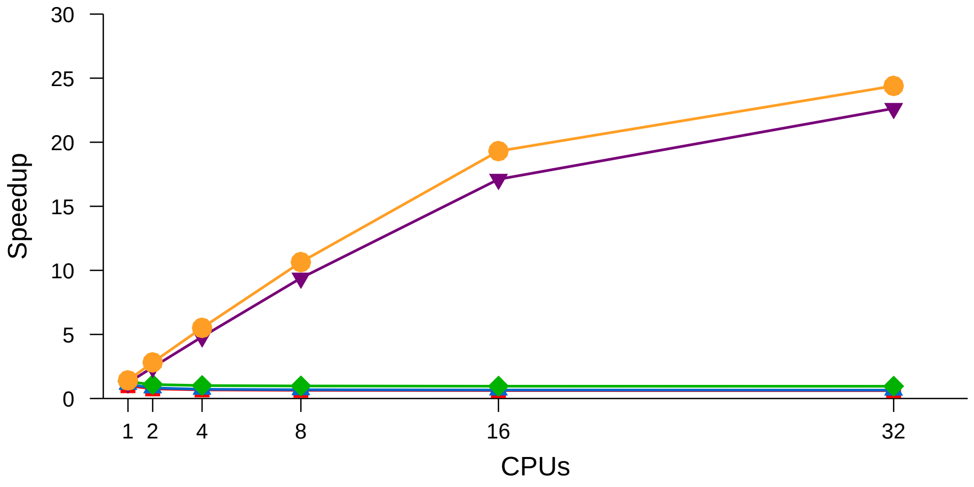
`HashMap` (HM) and `Hashtable` (HT) show the problems of scaling when using a simple critical section on traditional multiprocessors. Both `Hashtable` and the `synchronizedMap` version of `HashMap` actually slow down as more threads are added. While `ConcurrentHashMap` (CHM Fine) shows that fine-grained locking implementation is scalable up to 16 CPUs in this hardware design, this implementation requires significant complexity. With JavaT and transactions, TestHashtable can use the simple `HashMap` with the same critical region as `ConcurrentHashMap` and achieve similar scalability. Again scalability for 32 CPUs is limited by communication-to-computation ratio.

3.3.4 TestCompound

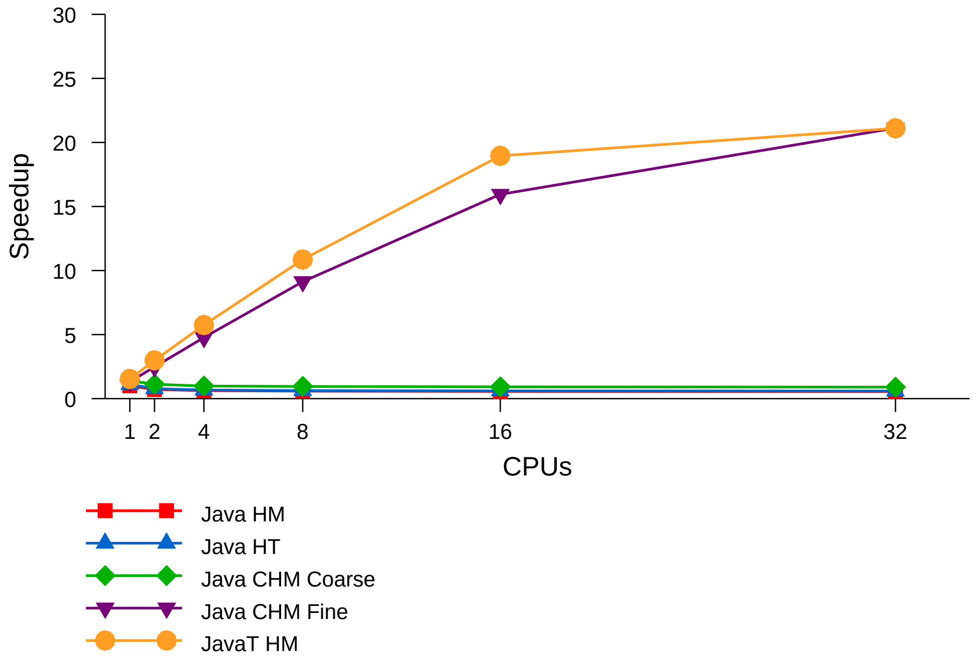
TestCompound is a micro-benchmark that compares the same `Map` implementations as TestHashtable. Again the threads contend for access to a single object instance, but this time instead of performing a single atomic operation on the shared instance, they need to perform four operations to swap the values of two keys. Two experiments are used to demonstrate both low-contention and high-contention scenarios: the low-contention case uses a 32k element table and the high-contention case uses a 8k element table.

The same basic `Map` implementations are used as before with TestHashtable. For `Hashtable` and `HashMap`, the critical section uses the `Map` instance as the lock. For `ConcurrentHashMap`, two variations of the benchmark are used representing coarse-grained locking and fine-grained locking. The coarse-grained locking variation uses the `Map` instance as a lock, as with `Hashtable` and `HashMap`. The fine-grained locking variation uses the keys of the values being swapped as locks, being careful to order the acquisition of the locks to avoid deadlock.

Figure 3.10a shows the results of TestCompound with low contention for locks. Again, running `HashMap` and `Hashtable` with a single lock show the problems of simple locking in traditional systems. Furthermore, the coarse-grained version of `ConcurrentHashMap` (CHM Coarse) demonstrates that simply getting programmers to



a.) TestCompound with low key contention



b.) TestCompound with high key contention

Figure 3.10: TestCompound performs four `Map` operations as a single compound operation to atomically swap the values of random keys. The low-contention version on the top uses 32k keys so there is low contention for any given key. The high-contention version on the bottom uses 8k keys. Results compare various `Map` implementations between 1–32 CPUs. Key: Java HM=`HashMap` with a single lock, Java HT=`Hashtable` with a single lock, Java CHM Coarse=`ConcurrentHashMap` with a single lock, Java CHM Fine=`ConcurrentHashMap` with fine-grained key locking, JavaT HM=`HashMap` with a single transaction.

use data structures designed for concurrent access is not sufficient to maximize application performance. With Java locking, the application programmer must understand how to use fine-grained locking in their own application to properly take advantage of such data structures. As the number of threads is increased, these applications with coarse-grained locks do not continue to scale, and as shown, often perform worse than the baseline case. Only fine-grained version of `ConcurrentHashMap` compares favorably with JavaT transactions. JavaT transactions have a performance advantage due to speculation. More importantly, JavaT transactions are able to beat the performance of fine-grained locks using only the most straightforward code consisting of a single `synchronized` statement and the unsynchronized `HashMap`. Hence, JavaT allows programmers to write simple code focused on correctness that performs better than complex code focused on performance.

Figure 3.10b shows the results of TestCompound with high contention for locks. The locking version of `Hashtable`, `HashMap`, and coarse-grained `ConcurrentHashMap` all perform similarly to the low-contention case. Fine-grained Java `ConcurrentHashMap` and JavaT transactional performance are both degraded from the low-contention case because of lock contention and data dependency violations.

3.3.5 SPECjbb2000

SPECjbb2000 is a server-side Java benchmark, focusing on business object manipulation. I/O is limited, with clients replaced by driver threads and database storage replaced with in-memory B-trees. The main loop iterates over five application transaction types: new orders, payments, order status, deliveries, and stock levels. New orders and payments are weighted to occur ten times more often than other transactions and the actual order of transactions is randomized. The default configuration that varies the number of threads and warehouses from 1 to 32 was used, although the measurement was for a fixed number of 736 application-level transactions instead of a fixed amount of wall clock time.

Figure 3.11 shows the results from SPECjbb2000. Both Java locking and JavaT transactional versions show linear speedup in all configurations because there is very

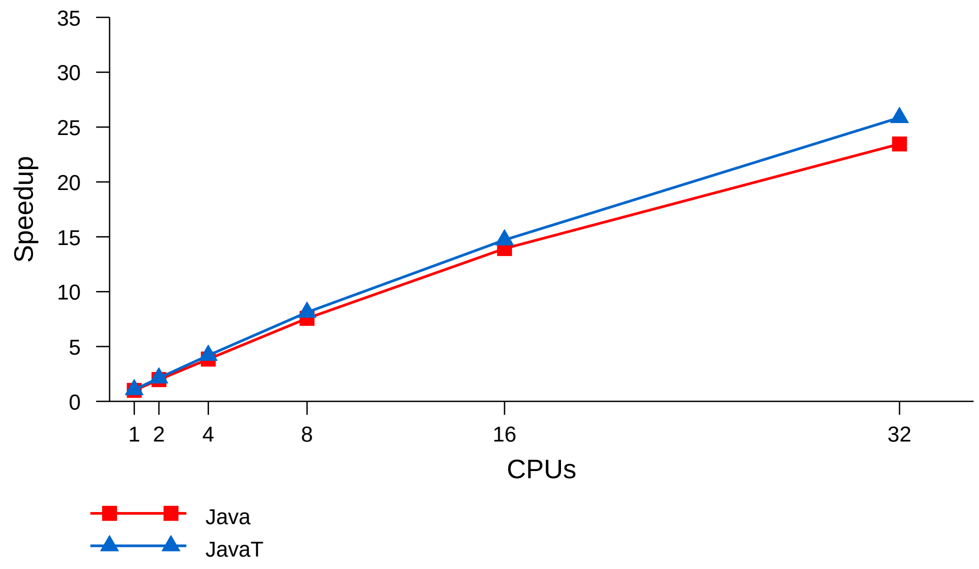


Figure 3.11: SPECjbb2000 results between 1–32 CPUs.

little contention between warehouse threads. However, the Java locking version is slightly slower in all cases because it must still pay the locking overhead to protect itself from the 1% chance of an inter-warehouse order. Most importantly, the JavaT transactional version of SPECjbb2000 did not need any manual changes to achieve these results; automatically changing `synchronized` statements to transactions and committing on `Object.wait()` was sufficient.

3.3.6 Java Grande

Java Grande provides a representative set of multithreaded kernels and applications. These benchmarks are often Java versions of benchmarks available in C or Fortran from suites such as SPEC CPU, SPLASH-2, and Linpack. The input sizes were shown previously in Table 3.2.

Figure 3.16–Figure 3.12 show the results from the Java Grande section 2 kernel programs. These highly-tuned kernels show comparable scaling when run with transactions. The transactional version of SOR is slightly slower than the lock version at

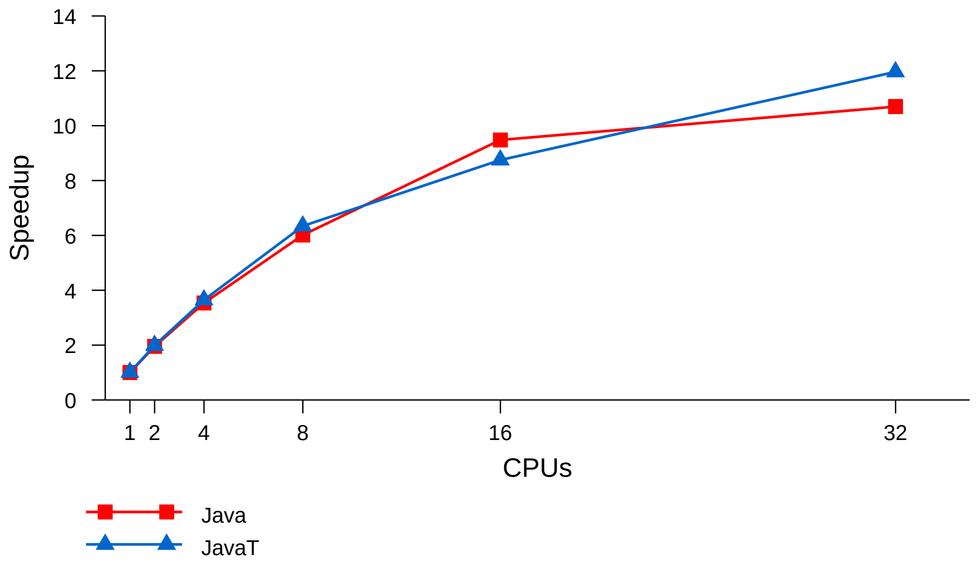


Figure 3.12: SOR Java Grande Forum section 2 kernel results between 1–32 CPUs.

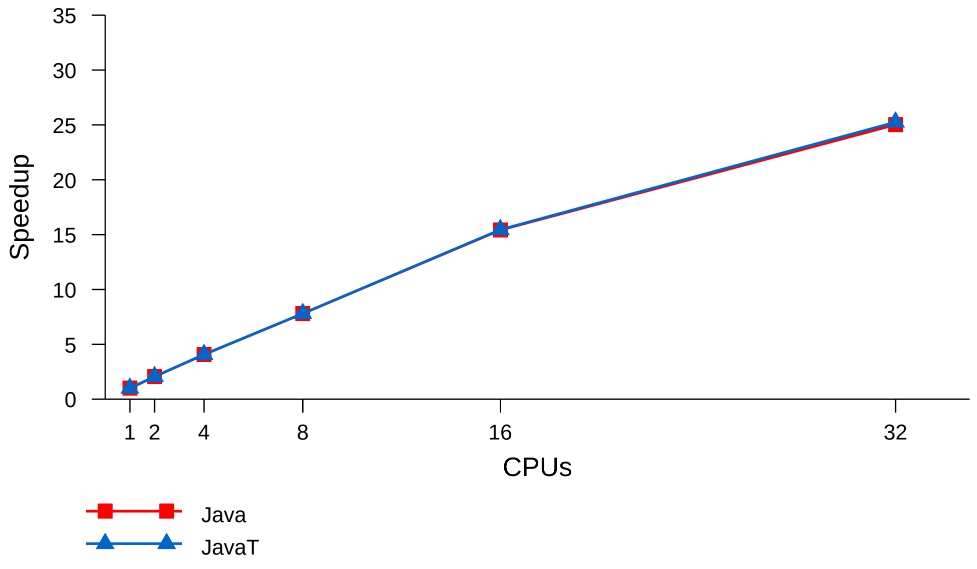


Figure 3.13: SparseMatmult Java Grande Forum section 2 kernel results between 1–32 CPUs.

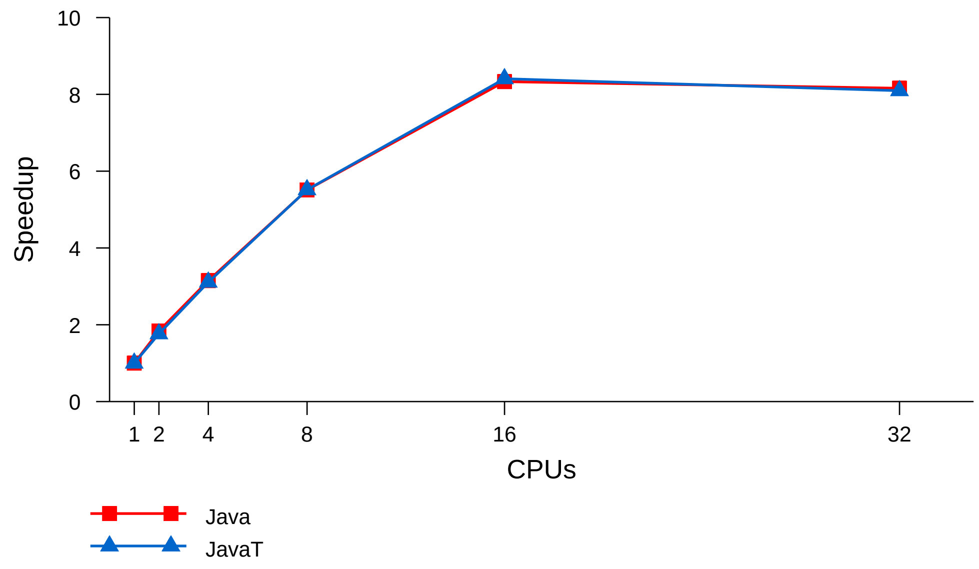


Figure 3.14: LUFact Java Grande Forum section 2 kernel results between 1–32 CPUs.

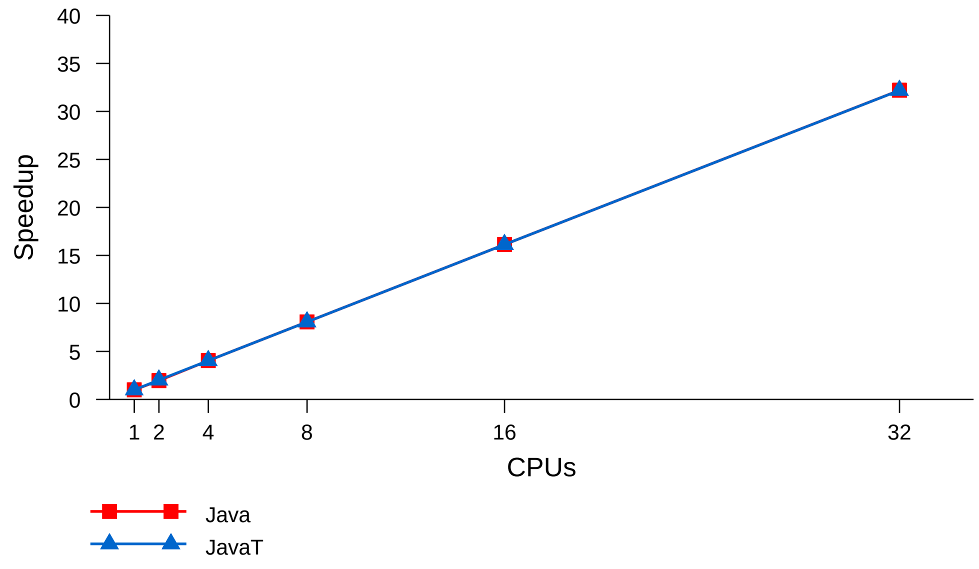


Figure 3.15: Crypt Java Grande Forum section 2 kernel results between 1–32 CPUs.

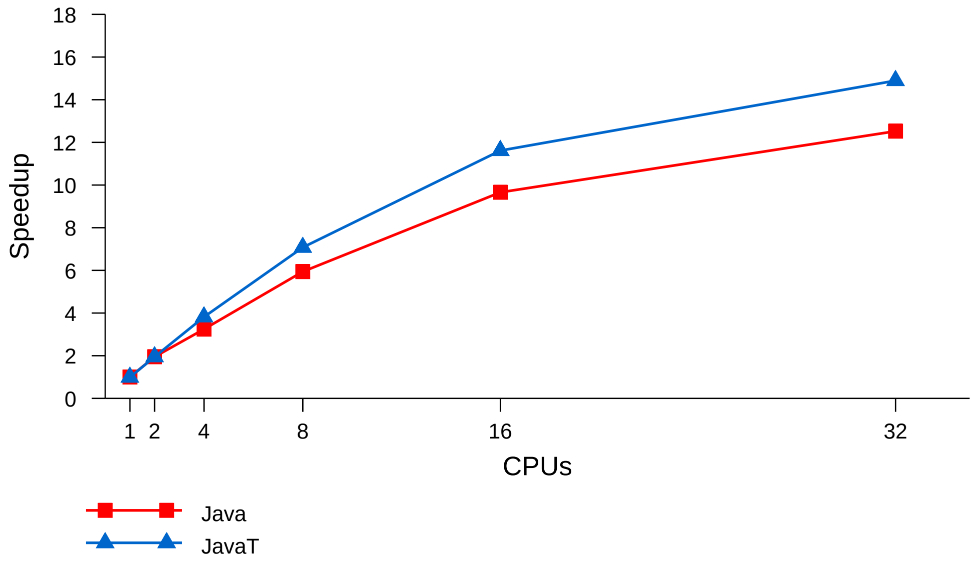


Figure 3.16: Series Java Grande Forum section 2 kernel results between 1–32 CPUs.

16 CPUs because of commit overhead, but as the number of threads increases, lock overhead becomes a factor for the version with locks. Transactions and locks perform equally well on SparseMatmult, LUFact, and Crypt, with Java lock overhead and violations having comparable cost as threads are added. The JavaT transactional version of Series has similar scaling to the lock-based version, but consistently higher performance due to the lower overhead of transactions compared to locks.

Figure 3.17–Figure 3.19 show the results from the Java Grande section 3 application programs. MolDyn has limited scalability for both Java locks and JavaT transactions, with a slightly higher peak speedup for transactions at 8 CPUs. MonteCarlo exhibits scaling with both Java locks and JavaT transactions, but with a slight performance edge for transactions at 32 CPUs. RayTracer is similar to the kernels SparseMatmult, LUFact, and Crypt, with nearly indentical performance for JavaT transactions and Java locks.

Three Java Grande benchmarks, LUFact, MolDyn, and RayTracer, use a common `TournamentBarrier` class that did not use `synchronized` statements but instead `volatile` variables. Each thread that enters the barrier performs a `volatile`

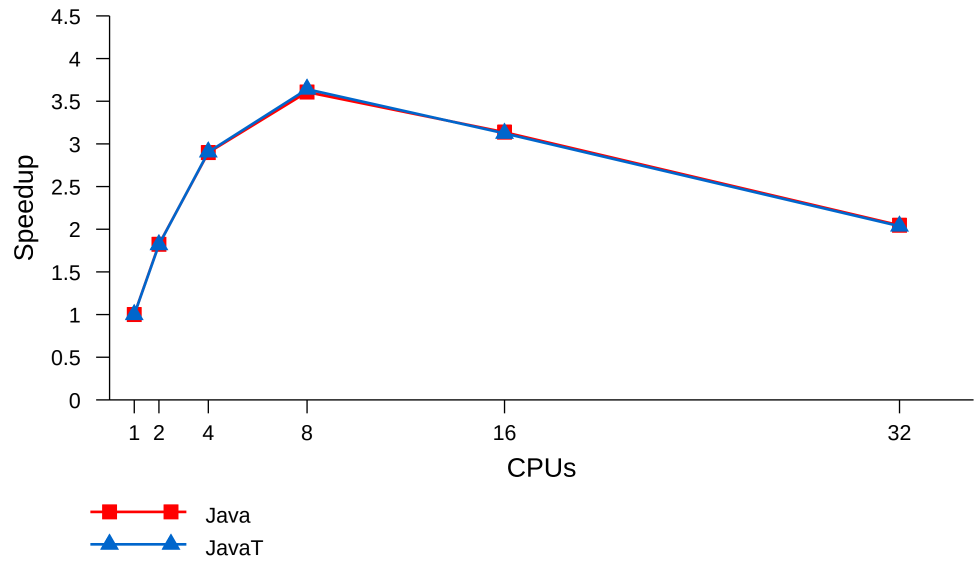


Figure 3.17: MolDyn Java Grande Forum section 3 application results between 1–32 CPUs.

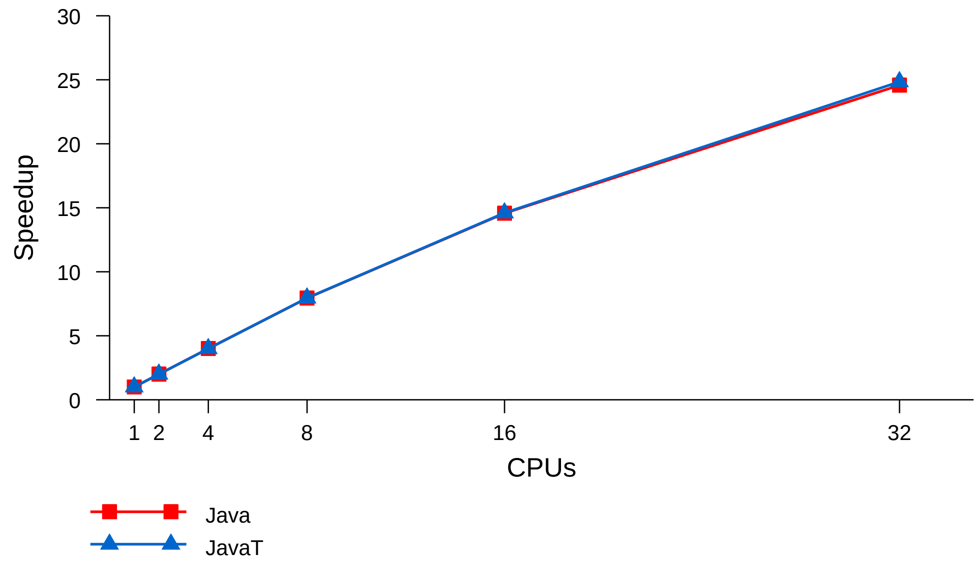


Figure 3.18: MonteCarlo Java Grande Forum section 3 application results between 1–32 CPUs.

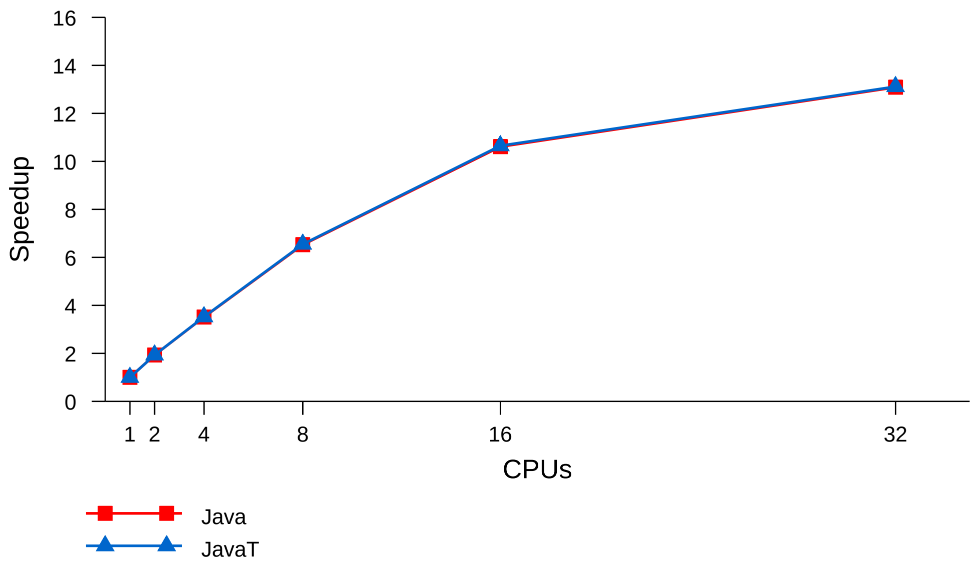


Figure 3.19: RayTracer Java Grande Forum section 3 application results between 1–32 CPUs.

write to indicate its progress and then busy waits to continue. Without the JavaT transactional interpretation of `volatile`, this write is not necessarily visible to other threads, and the usual result is that all threads loop forever, believing they are the only one to have reached the barrier. SOR had a similar usage of `volatile` variables that also worked correctly with the JavaT transactional rules.

3.4 Related Work

3.4.1 Speculating through Locks

JavaT's approach of converting lock regions into transactions is based on earlier work on lock speculation, which was covered earlier and in more detail in Section 2.5.3. Earlier TLS hardware based speculation, such as SLE, TLR, and Speculative Synchronization, had to detect updates to lock variables to avoid data dependency values on the lock implementation itself [105, 106, 87]. JavaT does not need to do this because the JIT compiler can emit HTM primitives for the `monitorenter` and `monitorenter` bytecodes. Azul's Java virtual machine follows a similar approach of using a combination of JIT and HTM support, although because they have to ensure Java semantics, they have to fall back to running with locks when condition variables are used [31].

Past studies trying to predict the behavior of transactional programs have focused converting lock-based critical sections to transactions. Ananian et al. studied sequential SPECjvm98 Java applications to try and understand the use of locks in existing programs [8]. Their results show that sequential Java programs hold locks for long periods of time, showing the need for HTM virtualization to support working sets larger than possible with on-chip caches. Chung et al. looked at a wider selection of Java, Pthread, and OpenMP applications from such benchmark suites as JavaGrande, SPECjbb2000, DaCapo, SPEComp, NAS, SPLASH, and SPLASH-2 [30]. These results found that in the common case the need for virtualization was rare, suggesting that software-based virtualization may be sufficient. However, in most cases these applications were already parallel, suggesting that they had already been optimized to minimize the use and length of critical sections.

3.4.2 Java with TM

Java has been a popular language for exploring transactional memory.

Herlihy et al. provided DSTM, a Java library based STM which requires Java objects to be explicitly made part of transactions [60]. DSTM2 updates this system with a more flexible factory based interface [59].

Welc et al. provide transactional monitors in Java through JikesRVM compiler extensions, treating `Object.wait()` as a thread yield without committing [131]. Unlike Welc et al., JavaT can run code containing conditional variables such as barriers that require communication on `Object.wait()`.

Saha et al. provide McRT, an STM runtime that can supports Java through the Intel's ORP virtual machine, as well as C and C++ programs through a library interface [112]. Adl-Tabatabai et al. later showed how an optimizing compiler can eliminate many unnecessary STM operations for greater performance [2].

Hindman and Grossman provided AtomJava as the first STM to provide strong isolation [62]. Later work by Shpeisman et al. showed how program analysis could significantly reduce the costs of providing strong isolation in an STM [118].

Grossman et al. explored potential issues in interactions between memory models such as Java's and transactional memory, covering issues beyond the simplified model from Section 3.2.1 [48].

3.4.3 Other Languages with TM

Harris et al. explored integrating software transactional memory with Concurrent Haskell [56]. Haskell is a mostly functional programming language where most memory operations do not need to be part of the rollback state.

Herlihy, who original worked on HTM and the DSTM library for Java mentioned above, also provided SXM, a Software Transactional Memory for the C# programming language [58].

Marathe et al. provided RSTM which is a library based STM for C++ [86]. Although later versions have improved the library interface in order to improve programability, a later paper notes the limitations of the library based approach [34].

Dice et al. provide the TL2 library for C++ [38]. It has been used in the evaluation of the STAMP transactional benchmark suite [122].

Ringenburg and Dan Grossman's AtomCaml is notable for its uniprocessor implementation that provides atomicity through rollback [109]. Kimball and Dan Grossman's AtomScheme explores the multiple possible semantics of integrating transactions with first-class continuations [73].

All three of DARPA's High-Productivity Computing System (HPCS) language proposals included support for an `atomic` construct for transactional memory. This includes Cray's Chapel [33], IBM's X10 [27], and Sun's Fortress [6].

3.5 Conclusion

JavaT provides some simple rules to evaluate transactional memory using existing Java programs. The microbenchmarks show that naive code can perform as well with transactions as specially designed code does with fine-grained locking. The Java-Grande and SPECjbb2000 results show that transactional memory can compete with locks on performance when translating fine-grained synchronization to transactions in well optimized Java programs. However, it was also shown that the pure transactional execution of Java programs can not be backward compatible due to lock-based condition variables. Chapter 4 will explore the Atomos transactional programming language that addresses this issue, as well as exploring the potential performance problems of larger transactions on both programming language design and implementation.

Chapter 4

The Atomos Transactional Programming Language

The devil is in the details.
– Anonymous

This chapter presents the Atomos transactional programming language, which was the first to include implicit transactions, strong isolation, and a scalable multiprocessor implementation. Atomos is derived from Java, but replaces its synchronization and conditional waiting constructs with transactional alternatives.

The Atomos conditional waiting proposal is tailored to allow efficient implementation with the limited transactional contexts provided by hardware transactional memory. There have been several proposals from the software transactional memory community for conditional waiting primitives that take advantage of transactional conflict detection for efficient wakeup [55, 56]. By allowing programmers more control to specify their conditional dependencies, Atomos allows the general ideas of these earlier proposals to be applied in both hardware and software transactional memory environments.

Atomos supports *open-nested* transactions, which were found necessary for building both scalable application programs and virtual machine implementations. Open nesting allows a nested transaction to commit before its parent transaction [88, 99]. This allows for parent transactions to be isolated from possible contention points in a more general way than other proposals like *early release*, which only allows a program to remove a location from its read-set to avoid violations [33].

4.1 Atomos = Java - Locks + TM

The Atomos programming language is derived from Java by replacing locking and conditional waiting with transactional alternatives. The basic transactional semantics are then extended through open nesting to allow programs to communicate between uncommitted transactions. Finally, commit and abort callbacks are provided so programs can either defer non-transactional operations until commit or provide compensating operations on abort.

4.1.1 Transactional Memory with Closed Nesting

Transactions are defined by an `atomic` statement similar to other transactional memory proposals [55, 27, 6, 33]. Because Atomos specifies strong isolation, statements within an `atomic` statement appear to have a serialization order with respect to other transactions as well as to reads and writes outside of transactions; reads outside of a transaction will not see any uncommitted data and writes outside a transaction can cause a transaction to roll back. Here is a simple example of an atomic update to a global counter:

```
atomic { counter++; }
```

Nested `atomic` statements follow closed-nesting semantics, meaning that the outermost `atomic` statement defines a transaction that subsumes the inner `atomic` statement. When a nested transaction commits, it merges its read- and write-sets with its parent transaction. When a transaction is violated, only it and its children need to be rolled back; the parent transaction can then restart the nested child.

The use of `atomic` statements conceptually replaces the use of `synchronized` statements. Studies show that this is what programmers usually mean by `synchronized` in Java applications [42]. However, this is not to say that programmers can blindly substitute one statement for the other, as it can affect the semantics of existing code [11]. Fortunately, this does not seem to be a common problem in practice [21, 20].

The concept of the `volatile` field modifier is replaced with an `atomic` field modifier as proposed in Chapel [33]. Fields marked with the `atomic` modifier are accessed as small closed-nested transactions that may be top-level or nested as specified above. This serves one purpose of `volatile`, since it restricts the reordering of access between top-level transactions. However, another purpose of transactions is to force visibility of updates between threads. This usage of `volatile` is discussed below in Section 4.1.3 on reducing isolation between transactions.

Atomos prohibits starting threads inside of transactions. This eliminates the semantic questions of nested parallelism where multiple threads could run within a single transactional context. Other proposals are also exploring such semantics, but there does not seem to be a consensus about the practical usefulness or the exact semantics of this feature [85, 36, 93].

4.1.2 Fine-Grained Conditional Waiting

Atomos conditional waiting follows the general approach of an efficient Conditional Critical Region (CCR) implementation: a program tests some condition, and finding it false, waits to restart until it has reason to believe the condition might now be true [55]. One nice semantic property of CCRs is that the waiting thread does not need to coordinate *a priori* with other threads that might be affecting the condition. Unfortunately, this lack of connection between waiting threads and modifying threads historically led to problematic performance for CCRs because either the condition was reevaluated too often, resulting in polling-like behavior, or not often enough, resulting in delayed wakeup. Transactional memory has made CCRs efficient by using the read-set of the transaction as a tripwire to detect when to reevaluate the condition; when a violation of the read-set is detected, another transaction has written a value that was read as part of evaluating the condition, so there is some reason to believe that the value of the condition has changed.

One problem with this approach is that it requires a transactional context with

```
public int get (){                     public int get() {
    synchronized (this) {                  atomic {
        while (!available)                     if (!available) {
            wait();                                watch available;
        available = false;                         retry;}
        notifyAll();                           available = false;
        return contents;}}                     return contents;}}
public void put(int value){            public void put (int value) {
    synchronized (this) {                  atomic {
        while (available)                      if (available) {
            wait();                                watch available;
        contents  = value;                         retry;}
        available = true;                      contents  = value;
        notifyAll();}}                         available = true;}
```

Figure 4.1: Comparison of producer-consumer in Java (left) and Atomos (right). The Java version has an explicit loop to retest the condition where the Atomos rollback on `retry` implicitly forces the retesting of the condition. Java requires an explicit notification for wakeup where Atomos relies on the violation detection of the underlying transactional memory system.

a potentially large read-set to remain active to listen for violations even while the thread has logically yielded. While this can scale well for software transactional memory systems that maintain such structures in memory, it is problematic for hardware transactional memory systems with a limited number of hardware transactional contexts, typically one per processor.

Atomos refines this general approach by allowing a program to specify a *watch-set* of locations of interest using the `watch` statement. After defining the watch-set, the program calls `retry` to roll back and yield the processor. The implementation of the `retry` statement communicates the watch-set to the scheduler, which then listens for violations on behalf of the now yielded thread. By rolling back, `retry` allows the waiting thread to truly yield both its processor resource and transactional context for use by other threads. Violation of the watch-set and restart of waiting threads are transparent to the writing thread, which does not need to be concerned about a separate explicit notification step as is necessary in most lock-based conditional waiting schemes.

Figure 4.1 shows a simple producer-consumer example derived from Sun's Java

```
synchronized (lock) {               atomic {
    count++;                            count++;}
    if (count != thread_count)      atomic {
        lock.wait();                    if (count != thread_count) {
    else                                    watch count;
        lock.notifyAll();}                  retry;}}
```

Figure 4.2: Comparison of a barrier in Java (left) and Atomos (right). `count` is initialized to zero for new barriers. The Java version implicitly has two critical regions since the `wait` call releases the monitor. In Atomos, the two critical regions are explicit.

Tutorial to compare and contrast Java's conditional variables with Atomos's `watch` and `retry` [18]. Although these two versions seem very similar, there are notable differences. First, note that Java requires an explicit association between a lock, in this case `this`, and the data it protects, in this case `available` and `contents`. Atomos transactions allow the program to simply declare what they want to appear atomic. Second, Java's `wait` releases the lock making side effects protected by the lock visible. Instead, Atomos's `retry` reverts the transaction back to a clean state. Third, because Atomos reverts back to the beginning of the transaction, the common Java mistake of using an `if` instead of a `while` in testing the condition is eliminated, since calling `retry` ensures that the program will always reevaluate the condition. Finally, Java requires an explicit notify to wake up waiting threads, where Atomos relies on implicit violation handling. This reduces the need for coordination and allows threads to wait for conditions without defining *a priori* conventions for notification.

To understand why this third difference is important, consider Figure 4.2 which shows an example derived from Java Grande's `SimpleBarrier` [67]. As noted above, Java `wait` will release the lock and make the updated `count` visible to other threads. However, replacing Java `wait` with Atomos `retry` would cause the code to roll back, losing the `count` update. All threads would then think they were first to the barrier and the program will hang. To create a transactional version, the code needs to be rewritten to use two transactions: one that updates the `count` and another that watches for other threads to reach the barrier. Code with side effects before conditional waiting is not uncommon in existing parallel code, as a similar example is given in Sun's Java Tutorial [18]. SPECjbb2000 also contains such an example with nested `synchronized` statements [121]. Fortunately, although they share a common pattern, such examples make up a very small fraction of the overall program and are generally easy to rewrite.

4.1.3 Open Nesting

Basic transactional behavior depends on the detection of conflicting read- and write-sets. However, in the world of databases, transactions often reduce their isolation from each other to gain better performance. Chapel [33] provides an early release construct, which is intended to prevent unwanted dependencies between transactions. Atomos takes a different approach by providing open nesting, which allows communication from within uncommitted transactions while still providing atomicity guarantees for updates made within the transaction.

The `open` statement allows a program to start an open-nested transaction. Where a closed-nested transaction merges its read- and write-set into its parent at commit time, an open-nested transaction commit always makes its changes globally visible immediately. For example, in Figure 4.3 when $\mathrm{T_E}$ commits, its updates are made visible to both other transactions in the system as well as its parent, $\mathrm{T_D}$. In contrast, when closed-nested transactions such as $\mathrm{T_B}$ and $\mathrm{T_C}$ commit, their changes are only made available to their parent, $\mathrm{T_A}$. Only when the parent $\mathrm{T_A}$ commits are changes from $\mathrm{T_B}$ and $\mathrm{T_C}$ made visible to the rest of the system. Like closed-nested transactions, the violation of an open-nested child does not roll back the parent, allowing the child to be resumed from its starting point, minimizing lost work.

Open-nesting semantics can seem surprising at first, since changes from the parent transaction can be committed if they are also written by the open-nested child transaction, seemingly breaking the atomicity guarantees of the parent transaction. However, in the common usage of open-nested transactions, the write-sets are typically disjoint. This can be enforced through standard object-oriented encapsulation techniques as discussed in Chapter 5.

To see how open-nested transactions are useful to application programs, consider

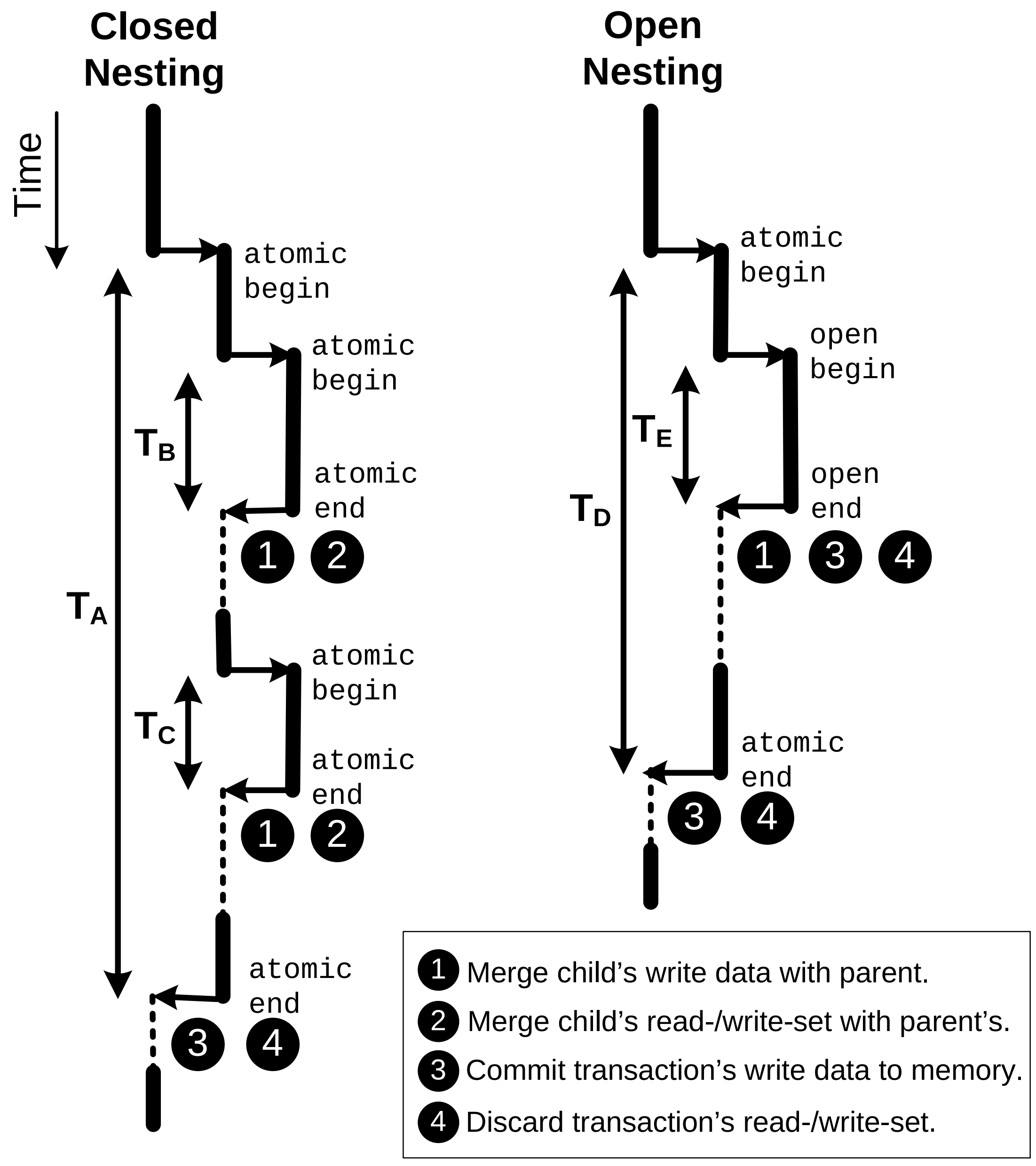

Closed Nesting
Open Nesting
Time
atomic begin
atomic begin
atomic end
1 2
T_A
T_B
atomic begin
T_C
atomic end
1 2
atomic end
3 4
atomic begin
open begin
T_E
T_D
open end
1 3 4
atomic end
3 4
1 Merge child's write data with parent.
2 Merge child's read-/write-set with parent's.
3 Commit transaction's write data to memory.
4 Discard transaction's read-/write-set.

Figure 4.3: Timeline of three nested transactions: two closed-nested and one open-nested. Merging of data (❶ and ❸) and read-/write-sets (❷ and ❹) is noted at the end of each transaction.

an example of generating unique identifiers for objects using a simple counter derived from SPECjbb2000 [121]. A program may want to create objects, obtain a unique identifier, and register the object atomically as sketched in the following code:

```
public static int generateID {
    atomic {
        return id++;}}
public static void createOrder (...) {
    atomic {
        Order order = new Order();
        order.setID(generateID());
        // finish initialization of order.
        // this could include creating more
        // objects which could mean more
        // calls to generateID.
        ...;
        orders.put(new Integer(order.getID()),
                   order);}}
```

However, doing so will mean that many otherwise unrelated transactions will have conflicting accesses to the `id` variable, even though the rest of their operations may be non-conflicting. By changing `generateID` to use open nesting, the counter can be read, updated, and committed quickly, with any violation rollback limited to the open-nested transaction, and not the parent application transactions:

```
public static open int generateID {
    open {
        return id++;}}
```

Open-nested transactions can also be used for more general inter-transaction communication like that found with transaction synchronizers [85].

Open-nested transactions allow threads to communicate between transactions while minimizing the risk of violations. Their use has some similarity to `volatile` variables in that commit forces writes to be immediately visible even before the parent transaction commits. For this reason, the `open` field modifier is allowed in cases where a `volatile` field was used within a `synchronized` statement.

Once an open-nested transaction has committed, a rollback of one of its parent transactions is independent from the completed open-nested transaction. If the parent is restarted, the same open-nested transaction may be rerun. In the unique identifier example, this is harmless as it just means there might be some gaps in the identifier sequence. In other cases, another operation might be necessary to compensate for the effects of the open-nested transaction, as discussed below in Section 4.1.4.

Open-nested transactions are also very useful in virtual machine implementation where runtime code runs implicitly within the context of a program transaction. For example, the JikesRVM JIT compiler adds its own runtime code to methods as it compiles them for several purposes: a.) code that checks for requests from the scheduler to yield the processor for other threads to run, b.) code that checks for requests from the garbage collector to yield the processor for the garbage collector to run, and c.) code that increments statistic counters, such as method invocation counters, that are used to guide adaptive recompilation. By using open-nested transactions, the runtime can check for these requests or increment these counters without causing the parent application transactions to roll back. Note that these three examples were benign data races in the original system, meaning that locks were not used to coordinate access to shared data. Such use of non-blocking synchronization can often result in unexpected violations. The use of open-nested transactions for virtual machine implementation will be discussed further in Section 4.2.1, which will cover how watch-sets are communicated to the scheduler.

4.1.4 Transaction Handlers

In database programming, it is common to run code based on the outcome of a transaction. Transactional-C provided `onCommit` and `onAbort` handlers as part of the language syntax. Harris extended this notion to transactional memory with the `ContextListener` interface [54]. Harris introduces the notion of an `ExternalAction`, which can write state out of a transactional context using Java `Serialization` so that abort handlers can access state from the aborted transaction.

In Atomos, open-nested transactions fill the role of `ExternalAction` by providing a way to communicate state out of a transaction that might later be needed after rollback. Separate `CommitHandler` and `AbortHandler` interfaces are provided so that one or the other may be registered independently:

```
public interface CommitHandler {
    public void onCommit();}
public interface AbortHandler {
    public void onAbort();}
```

Each level of nested transactions can register handlers. When registered, a handler is associated with the outermost nested transaction and is run at its conclusion in a new transactional context. Commit handlers run in a closed-nested transaction and abort handlers run in an open-nested transaction. This supports semantic concurrency control as described in Chapter 5. See Section 5.3 for more details. Atomos runs commit handlers in the second half a two phase commit sequence to support non-transactional operations as described in Section C.4.

In database programming, transaction handlers are often used to integrate non-transactional operations. For example, if a file is uploaded to a temporary location, on commit it would be moved to a permanent location and on abort it would be deleted. In transactional memory programming, transaction handlers serve similar purposes. Transaction handlers can be used to buffer output or rewind input performed within transactions. Transaction handlers can be used to provide compensation for open-nested transactions. In the JIT example, 100% accurate counters were not required. If a method is marked as invoked and then rolls back, it is not necessary to decrement the counter. Indeed, as mentioned before, there was no synchronization in the non-transactional JikesRVM execution to provide 100% accuracy. However, programs such as the SPECjbb2000 benchmark that keep global counters of allocated and deallocated objects want accurate results. An abort handler can be used to compensate the open transaction, should a parent transaction abort. Further details on handlers, including

```
public final class Transaction {
  public static Transaction getCurrentTransaction();
  public static void registerCommitHandler(CommitHandler c);
  public static void registerAbortHandler(AbortHandler a);
  public void abort();
}
```

Figure 4.4: The `Transaction` class used for handler registration and program directed transaction abort.

an I/O example, can be found in [88].

4.1.5 Transaction Class

The Atomos `Transaction` is shown in Figure 4.4. The class supports the registration of commit and abort handlers via the `registerCommitHandler` and `register-AbortHandler` methods as described above. The `Transaction` class also provides a `getCurrentTransaction` method that returns a `Transaction` instance that can be used with the `abort` method to roll back a transaction. As discussed in Chapter 5, transactions may need to be aborted by other threads, so references to `Transaction` instances may be passed between threads.

4.2 Implementing Transactional Features

This section will detail two transactional programming constructs implemented in Atomos. The first is a discussion of the implementation of Atomos `watch` and `retry`, which demonstrates a use of open-nested transactions and violation handlers. The second is a demonstration of how simple TLS-like loop speculation can be built with closed-nested transactions.

4.2.1 Implementing retry

Implementing violation-driven conditional waiting with hardware transactional memory is challenging because of the limited number of hardware transactional contexts for violation detection, as mentioned previously in Section 4.1.2. The problem is making sure someone is listening for the violation even after the waiting thread has yielded. The Atomos solution uses the existing scheduler thread of the underlying JikesRVM implementation to listen for violations on behalf of waiting threads.

The Atomos implementation uses a *violation handler* to communicate the watch-set between the thread and the scheduler. A violation handler is a callback that allows a program to recover from violations instead of necessarily rolling back. Violation handlers are a more general form of abort handlers that allow complete control over what happens at a violation including whether or not to roll back a specific nesting level, where to resume after the handler, and what program state will be available after the handler. Violation handlers run with violations blocked by default, allowing a handler to focus on handling the violations for a specific nesting level without having to worry about handler re-entrance. If a violation handler chooses to roll back a violated transaction at a specific nesting level, any pending violations for that nesting level are discarded. Otherwise, the violation handler will be invoked again with the pending violation after it completes.

Violation handlers are not a part of the Atomos programming language. They are the mechanism used to implement the higher-level Atomos `AbortHandler`, which is not allowed to prevent roll back. At the operating system level, violation handlers are implemented as synchronous signal handlers that run in the context of the violated transaction. The handler can choose where to resume the transaction or to roll back the transaction. The signal handler is integrated into the Atomos runtime via the higher level `ViolationHandler` interface that is more restrictive than the general purpose signal handler:

```
public interface ViolationHandler {
    public boolean onViolation(
        Address violatedAddress);}
```

This restricted form of handler can either return true if it handled the violation and wants to simply resume the interrupted transaction or return false if it wants the transaction to roll back. The underlying lower-level violation signal handler takes

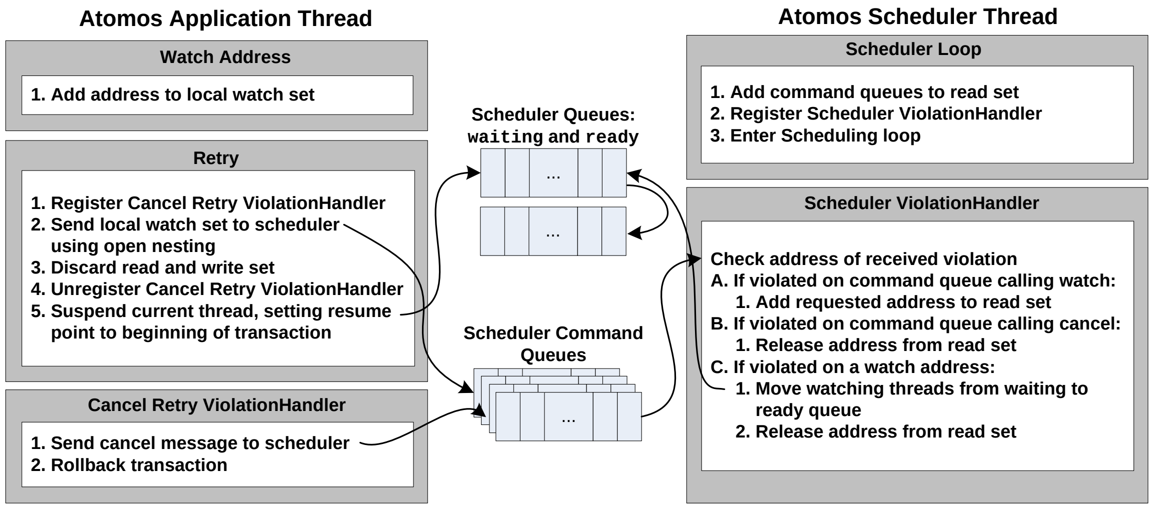

Atomos Application Thread
Watch Address
1. Add address to local watch set
Retry
1. Register Cancel Retry ViolationHandler
2. Send local watch set to scheduler using open nesting
3. Discard read and write set
4. Unregister Cancel Retry ViolationHandler
5. Suspend current thread, setting resume point to beginning of transaction
Cancel Retry ViolationHandler
1. Send cancel message to scheduler
2. Rollback transaction
Scheduler Queues: waiting and ready
Scheduler Command Queues
Atomos Scheduler Thread
Scheduler Loop
1. Add command queues to read set
2. Register Scheduler ViolationHandler
3. Enter Scheduling loop
Scheduler ViolationHandler
Check address of received violation
A. If violated on command queue calling watch:
1. Add requested address to read set
B. If violated on command queue calling cancel:
1. Release address from read set
C. If violated on a watch address:
1. Move watching threads from waiting to ready queue
2. Release address from read set

Figure 4.5: Conditional synchronization using open nesting and violation handlers. Waiting threads communicate their watch-sets to the scheduler thread via scheduler command queues in shared memory that interrupt the scheduler loop using violations. The Cancel Retry ViolationHandler allows the waiting thread to perform a compensating transaction to undo the effects of its open-nested transactions in the event of rollback of the parent transactions.

care of calculating the address of where to resume.

Figure 4.5 sketches out the basic implementation of Atomos conditional waiting. The `watch` statement simply adds an address to a thread-local watch-set. The `retry` implementation uses `open` to send this watch-set to the scheduler thread, which is effectively listening for requests from the other threads using a violation handler. Once the thread is sure that the scheduler is listening on its watch-set, it can roll back and yield the processor. The scheduler's violation handler serves three purposes. First, it watches for requests to read watch-set addresses. Second, it handles requests to cancel watch requests when the thread is violated in the process of waiting. Finally, it handles violations to watch-set addresses by ensuring that watching threads are rescheduled.

To illustrate how this works, consider the example of the producer-consumer code in Figure 4.1. Suppose a consumer thread finds that there is no data available. It requests to watch `available`, which simply adds the address of `available` to a local list. When the thread calls `retry`, the thread uses an open-nested transaction to send the `available` address to the scheduler, which then reads it. The scheduler then uses its own open-nested transaction to acknowledge that it has read the address, so that the original thread can now roll back and yield. Later on, a producer makes some data available. Its write to `available` causes the scheduler to be violated. The scheduler finds that it had read the address of `available` on behalf of the original consumer thread, which it then reschedules for execution.

Figure 4.6 and Figure 4.7 sketch the code run by the waiting thread and the scheduler thread, corresponding to the left and right of Figure 4.5. The `watch` implementation simply adds the address to the thread local `watchSet` list. Note that the address never needs to be explicitly removed from the watch-set because this transaction will be rolled back either by a violation from another thread or when the thread is suspending.

The `retry` implementation needs to communicate the addresses from the `watchSet` to the scheduler thread so it can receive violations on behalf of the waiting thread after it suspends. To do this, the waiting thread uses open-nested transactions. However, the transaction could be violated while communicating its watch-set. The scheduler would then be watching addresses for a thread that has already been rolled back. In order to keep consistent state between the two threads, the waiting thread uses a violation handler to perform a compensating transaction to let the scheduler know to undo the previous effects of the waiting thread's open-nested transactions. In order to achieve this, it is important to register the `retryVH` violation handler before any communication with the scheduler. This violation handler is only unregistered after the thread's `watchSet` has been communicated and the read-set of the waiting thread has been discarded to prevent violations.

In this example implementation of `retry`, `schedulerAddress` is used to communicate `watchSet` addresses to the scheduler and `schedulerWatch` is set to true to violate the scheduler thread, invoking its violation handler. After the first `open` transaction, a second `open` is used to listen for an acknowledgment from the scheduler so that the waiting thread has confirmation of the `watchSet` transfer before discarding state, ensuring that at least one of the two threads will receive the desired violations at any time.

```
// watch keyword adds an address to local wait set
void watch(Address a){
  VM_Thread.getCurrentThread().watchSet.add(a); }

// retry keyword implementation
void retry(){
  VM_Thread thread = VM_Thread.getCurrentThread();
  List watchSet = thread.watchSet;
  // register "cancel retry violation handler" to
  // cleanup scheduler if thread violates before yield
  VM_Magic.registerViolationHandler(retryVH);
  for (int i=0,s=watchSet.size();i<s;i++){
      Address a=(Address)watchSet.get(i);
      open {
          // write address where scheduler is reading
          thread.schedulerAddress = a;
          // wakeup the scheduler violation handler
          thread.schedulerWatch = true; }
      // busy wait until thread hears from scheduler
      open { if (thread.scheduleWatch) for(;;) ; }}
  // clear the thread's read-set to avoid violations
  // now that scheduler is listening for us
  VM_Magic.discardState();
  // safe to unregister now that read-set cleared
  VM_Magic.unregisterViolationHandler(retryVH);
  // store resume context from checkpoint and yield
  thread.suspend(); }

// cancel retry violation handler (retryVH)
boolean onViolation(Address a){
  VM_Thread thread = VM_Thread.getCurrentThread();
    open {
      thread.schedulerCancel = true; }
    open { if (thread.schedulerCancel) for(;;) ; }
  return false; } // rollback transaction
```

Figure 4.6: Implementation details of Atomos `watch` and `retry` using violation handlers. This code is run by the waiting thread, shown on the left side of Figure 4.5.

```
// Scheduler violation handler
boolean onViolation(Address a){
  VM_Thread t = schedulerWatches.get(a);
  if (t != null) { // case A: watch request
    Address address;
    // read the next watch address
    open { address = t.schedulerAddress; }
    address.loadWord(); // load adds to read-set
    open {
      // update address and thread mappings for below
      addressToThreads.get(address).add(t);
      threadToAddresses.get(t).add(address);
      // let the sender continue
      thread.schedulerWatch = false;
      return true; }} // never rollback transaction
  t = schedulerCancels.get(a);
  if (t != null) { // case B: retry cancel request
    open {
      List addresses = threadToAddresses.remove(t);
      for (int j=0, sj=addresses.size();j<sj;j++){
        Address a = (VM_Thread)addresses.get(j);
        Map threads = addressToThreads.get(address);
        threads.remove(t);
        if (threads.isEmpty()) {
            VM_Magic.releaseAddress(a); }}
      thread.schedulerCancel = false;
      return true; }} // never rollback transaction
  open { // notification for some thread?
    List threads = addressToThreads.remove(a);
    if (threads != null) { // case C: resume threads
      for (int i=0, si=threads.size();i<si;i++){
        VM_Thread t = (VM_Thread)threads.get(i);
        t.resume();
        List addresses = threadToAddresses.remove(t);
        for (int j=0, sj=addresses.size();j<sj;j++){
          Address a = (Address)threads.get(j);
          Map moreThreads = addressToThreads.get(a);
          moreThreads.remove(t);
          if (moreThreads.isEmpty()) {
              VM_Magic.releaseAddress(a); }}}}}
  return true; }  // never rollback transaction
```

Figure 4.7: Implementation details of Atomos `watch` and `retry` using violation handlers. This code is run by the scheduler thread, shown on the right side of Figure 4.5. The Scheduler ViolationHandler cases A, B, and C from Figure 4.5 are marked with comments in the Scheduler `onViolation` code.

Below the `retry` code is `retryVH`, the violation handler for the waiting thread. It uses the same technique to communicate with the scheduler. The violation handler returns false to indicate that the waiting thread should be rolled back, allowing the waiting thread to reevaluate its wait condition.

Figure 4.7 shows the `onViolation` code for the scheduler thread. It uses the `schedulerWatches` map to determine if this is a watch request from a waiting thread. The `schedulerWatches` and related `schedulerCancels` maps are established when the `VM_Thread` objects are created during virtual machine initialization; programming language threads are multiplexed over the `VM_Thread` instances. If the violation is from a known `schedulerWatch` address, the value in `schedulerAddress` field is added to the read-set of the scheduler simply by loading from the address. The `schedulerAddress` value is read in an open-nested transaction to avoid adding the location of this field to the scheduler read-set. The thread and address information is then used to update `addressToThreads` and `threadToAddresses` maps. The `addressToThreads` is used when a violation is received to determine the threads that have requested wakeup. The `threadToAddresses` map is used to track addresses to remove from the scheduler read-set when there is a cancel request.

If the violation is instead from a `schedulerCancel` address, the scheduler needs to remove from its read-set any addresses that it was watching solely for the requesting thread, being careful to remove the location only if it is not in the watch-set of any other thread.

The final case in the scheduler code is to resume threads on a watch-set violation. After resuming, the threads will then reevaluate their conditional waiting code. In addition, the watch-sets of the resumed threads are removed from the scheduler read-set as necessary, similar to the code in the cancel case.

The scheduler thread must be very careful in managing its read-set or it will miss violations on behalf of other threads. The violation handler uses exclusively open-nested transactions to update `addressToThreads` and `threadToAddresses` and `schedulerCommand`. The scheduler main loop must also use only open-nested transactions as committing a closed-nested transaction will empty the carefully constructed read-set. Fortunately, such complex uses of open nesting are generally confined to

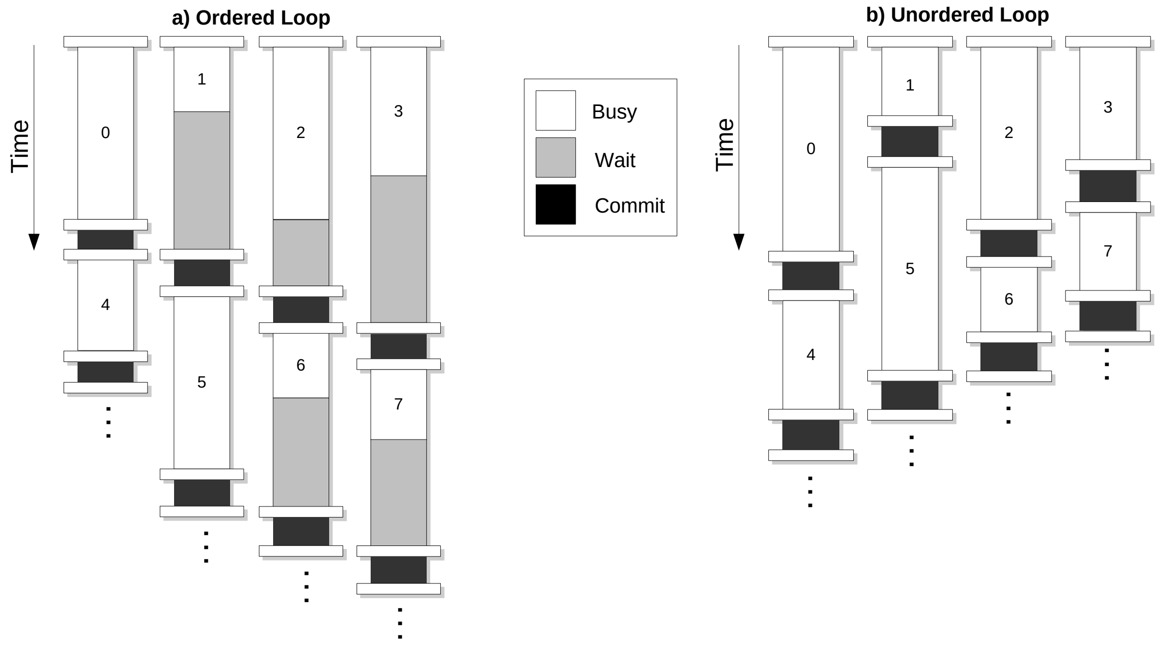


Figure 4.8: Ordered versus Unordered speculative loop execution timeline. The numbers indicate the loop iteration from sequential execution. In the ordered case on the left, iterations wait to commit to preserve sequential semantics, setting up a cascading commit pattern that often reduces the need for waiting in later iterations. In the unordered case on the right, iterations are free to commit in first come, first served order.

runtime system implementation and application uses are more straightforward as shown in the previous counter example.

4.2.2 Loop Speculation

The `t_for` loop speculation statement allows sequential loop-based code to be quickly converted to use transactional parallelism [51]. When used in ordered mode, it allows sequential program semantics to be preserved while running loop iterations in parallel. It also allows multiple loop iterations to be run in larger transactions to help reduce potential overheads caused by many small transactions, similar to loop unrolling.

Figure 4.8 gives a schematic view of running loop iterations in parallel using `t_for`. In Figure 4.8a, the sequential order is preserved by requiring that loop iterations

```
void run (boolean ordered, int chunk, List list, LoopBody loopBody){
    Thread[]  threads  = new Thread[cpus];
   // keep track as iterations finish
    boolean[] finished = new boolean[list.size()];
    for(int t=0; t<cpus; t++){
        threads[t] = new Thread(new Runnable(){
            public void run(){
                for(int i = t*chunk; i < list.size(); i+= cpus*chunk){
                    // run each chunk atomically
                    atomic {
                        for(int c=0; c<chunk; c++){
                            int iteration = i+c;
                            loopBody.run(list.get(iteration));
                            // mark when iterations complete
                            finished[iteration]=true; }
                        if (ordered){
                            if (i>0)
                                atomic {
                                    if (!finished[i-1])
                                        // preserve ordering by stalling
                                        // commit. when the previous
                                        // iteration updates finish,
                                        // restart the inner atomic
                                        for (;;) {
                                            ; }}}}}}});
        threads[t].start();}
    for(int t=0; t<cpus; t++){
        threads[t].join();}}
```

Figure 4.9: `Loop.run` implementation. For the unordered case, iterations simply run and commit as fast as possible. In the ordered case, iterations have to stall commit until the previous transaction finishes without losing the work of the loop body. This is performed by stalling in a nested `atomic` block at the end of loop iterations.

commit in order, even when iteration length varies. Note that although the commit of iterations has to be delayed to preserve sequential order at the start of the `t_for` loop, the resulting pattern of staggered transactions typically leads to a pipelining effect where future iterations do not have to be delayed before committing. In Figure 4.8b, iterations are free to commit in any order, removing the wait time shown in the ordering case.

In Atomos, statements like `t_for` can be implemented as library routines. For example, `t_for` can be recast as a `Loop.run` method that takes a `LoopBody` interface similar to the use of `Thread` and `Runnable`:

```
public class Loop {
    public static void run (
        boolean  ordered,
        int      chunk,
        List     list,
        LoopBody loopBody);}
public interface LoopBody {
    public void run (Object o);}
```

The `ordered` argument allows the caller to specify sequential program order if needed, otherwise transactions commit in "first come, first served" order. The `chunk` specifies how many loop iterations to run per transaction. The `list` argument specifies the data to iterate over. The `loopBody` specifies the method to apply to each loop element. Before the discussion of the `Loop.run` implementation, consider an example of using loop speculation. Consider the following `histogram` example program:

```
void histogram(int[] A,int[] bin){
    for(int i=0; i<A.length; i++){
        bin[A[i]]++;}}
```

The `Loop.run` routine can be used to convert `histogram` into the following parallel program:

```
void histogram(int[] A,int[] bin){
```

```
Loop.run(false,20,Arrays.asList(A),new LoopBody(){
   public void run(Object o){
     bin[A[((Integer)o).intValue()]]++;}}}
```

This version of `histogram` runs loop chunks of 20 loop iterations in parallel as unordered transactions.

Figure 4.9 shows a simple implementation of `Loop.run`. The unordered case is relatively straightforward with various chunks running atomically in parallel with each other. The ordered case is more interesting in its use of a loop within a closed-nested transaction to stall the commit until previous iterations have completed without rolling back the work of the current iterations. This general pattern can be used to build arbitrary, application-specific ordering patterns.

4.3 Evaluation of Atomos

This section evaluates the performance of Atomos transactional programming. Compared to the evaluation of JavaT in Section 3.3, this section will focus on evaluating the performance of Atomos transaction handlers and conditional waiting.

4.3.1 Handler Overhead in SPECjbb2000

An Atomos version of SPECjbb2000 was created by replacing `synchronized` blocks without conditional waiting with `atomic` blocks. The use of Java conditional waiting for creating barriers was replaced with Atomos code as shown in Figure 4.2. Although a barrier is used to synchronize the start and stop of measurement across all threads, the primary source of differences in execution time between JavaT and Atomos is the overhead of transactional handlers. Although no program defined handlers are used in this version of SPECjbb2000, there still is extra code to run `CommitHandlers` at the end of top-level `atomic` blocks as well as to run `AbortHandlers` on violations. Figure 4.10 shows the results of running SPECjbb2000 in three ways. The Java and JavaT results from Section 3.3.5 are shown in comparison to the Atomos result. As

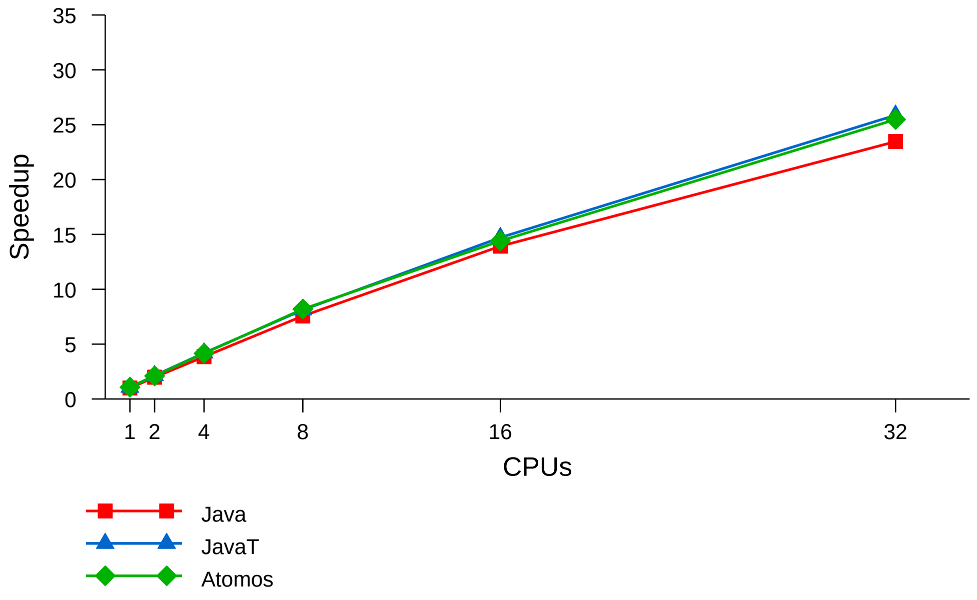


Figure 4.10: Comparison of the speedup of the Atomos version of SPECjbb2000 with the Java and JavaT versions. The Atomos version shows linear scaling up to 32 CPUs, matching the Java and JavaT versions.

you can seen, the overhead for transactional handlers is minor compared to the overall execution time of SPECjbb2000.

4.3.2 Conditional Waiting in TestWait

One of the contributions of Atomos is fine-grained conditional waiting. The Atomos implementation tries to minimize the number of transactional contexts required to support this and still achieve good performance. We present the results of a micro-benchmark that shows that the Atomos implementation does not adversely impact the performance of applications that make use of conditional waiting.

TestWait is a micro-benchmark that focuses on producer-consumer performance [55]. 32 threads simultaneously operate on 32 shared queues. The queues are logically arranged in a ring. Each thread references two adjacent queues in this ring, treating one as an input queue and one as an output queue. Each thread repeatedly attempts to read a token from its input queue and place it in its output queue. The queue implementation was based on `BoundedBuffer` from `util.concurrent` for Java and a `TransactionalBoundedBuffer` modified to use `watch` and `retry`. The experiment varies the number of *tokens*, not processors, from 1 to 32.

Figure 4.11 shows the results from TestWait. As the number of tokens increases, both Atomos and Java show similar speedups from 1 to 4 tokens, since both are paying similar costs for suspending and resuming threads. However, as the number of tokens approaches the number of processors something interesting happens. Recall that threads that are in the process of waiting but have not yet discarded their read-set can be violated and rescheduled without paying the cost of the thread switch. Up until the point that the read-set is discarded, a violation handler on the thread that has entered `retry` can cancel the process and simply restart the transaction without involving the scheduler. At 8 tokens, one quarter of the 32 processors have tokens at a time, so its very likely that even if a processor does not have a token it might arrive while it is executing `watch` or the watch request part of `retry`, allowing it to roll back and restart very quickly. At 16 tokens, this scenario becomes even more likely. At 32 tokens, this scenario becomes the norm. In the Java version, the mutual exclusion of

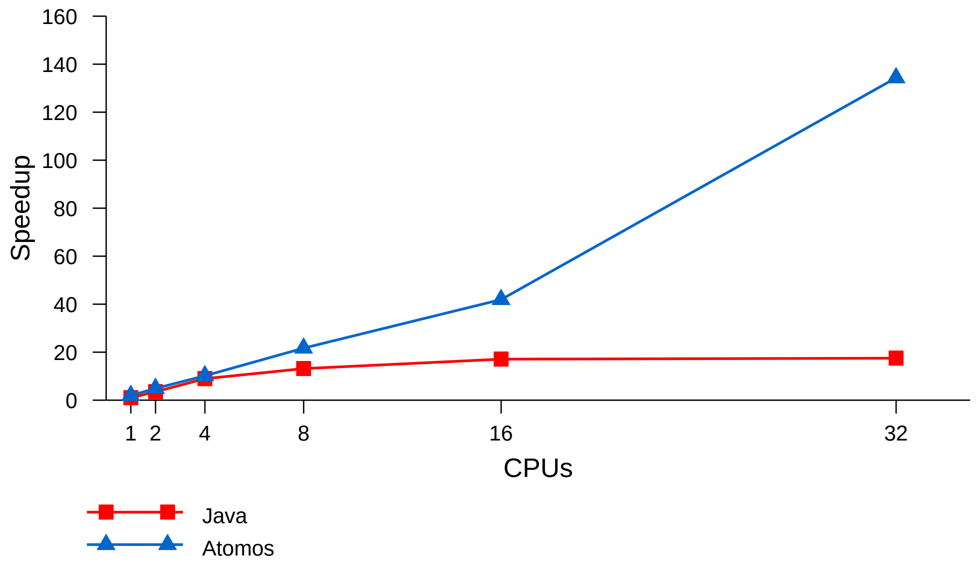


Figure 4.11: TestWait results comparing Java and Atomos conditional waiting implementation through a token passing experiment run in all cases on 32 CPUs. As the number of simultaneously passed tokens increases, both Java and Atomos take less time to complete a fixed number of token passes. The Atomos performance starts to accelerate with 8 tokens being passed on 32 CPUs when Cancel Retry ViolationHandler from Figure 4.5 frequently prevents the thread waiting code from completing a context switch. When 32 tokens are passed between 32 CPUs, there is almost no chance that the Atomos version will have to perform a context switch.

the monitor keeps the condition from changing while being tested, meaning that if the condition fails, the thread will now wait, paying the full cost of switching to the wait queue and back to running, even if the thread that could satisfy the condition is blocked waiting at the monitor at the time of wait.

4.4 Related Work

4.4.1 Programming Languages with Durable Transactions

Prior work on integrating database transactions with programming languages has provided inspiration for features such as transaction handlers. ARGUS was a programming language and system that used the concepts of guardians and actions to build distributed applications [84]. The Encina Transactional-C language provided syntax for building distributed systems including nested transactions, commit and abort handlers, as well as a "prepare" callback for participating in two-phase commit protocols [116]. Encina credits many of its ideas to work on Camelot and its Avalon programming language [41]. The SQL standards define how SQL can be embedded into a variety of programming languages [64]. There are also systems such as PJama that provide orthogonal persistence allowing objects to be saved transparently without explicit database code [70].

4.4.2 Open Nesting

The idea of open-nested transactions can be traced back to work by Gray [46]. Trager later coined the name *open nested* and *closed nested* to contrast Gray's work with Moss's work on nested transactions [124, 95]. Section 5.1.1 will provide a more detailed history of open-nested transactions in the context of semantic concurrency in databases.

Open nested transactions are part of several TM proposals for both hardware [99, 97, 94] and software [101]. Agarwal et al. provides several varieties of open nested semantics for transactional memory that include comparing various concrete proposals to the more formal semantics [5].

Name	**Implicit Transactions**	**Strong Atomicity**	**Programming Language**	**Multi-Processor**
Knight [74]	Yes	Yes	No	Yes
Herlihy & Moss [61]	No	No	No	Yes
TCC [53, 51]	Yes	Yes	No	Yes
UTM [8]	Yes	No	No	Yes
LTM [8]	Yes	Yes	No	Yes
VTM [107]	Yes	Yes	No	Yes
Shavit & Touitou [115]	No	No	No	Yes
Herlihy et al. [60]	No	No	No	Yes
Harris & Fraser [55]	Yes	No	Yes	Yes
Welc et al. [131]	Yes	No	Yes	Yes
Harris et al. [56]	Yes	Yes	Yes	No
AtomCaml [109]	Yes	Yes	Yes	No
X10 [27]	Yes	No	Yes	Yes
Fortress [6]	Yes	No	Yes	Yes
Chapel [33]	Yes	No	Yes	Yes
McRT-STM [112, 2]	Yes	No	Yes	Yes
Atomos	Yes	Yes	Yes	Yes

Table 4.1: Summary of transactional memory systems. The first section lists hardware transactional memory systems. The second section lists software transactional memory systems.

As an alternative to open nesting, Zilles and Baugh propose pausing and resuming transactions to allow execution outside the scope of the current transaction, requiring a system that mixes transactions with traditional mechanisms such as locks to coordinate within the non-transactional region, rather than a pure transactional model [133]. Section 5.1.3 will provide more details of how this alternative to open nesting can be used to support semantic concurrency control.

4.4.3 Atomos Compared to Other TM Systems

Table 4.1 summarizes how various STM and HTM proposals from Section 2.5 compare to Atomos using transactional memory properties from Chapter 2, showing that Atomos was the first transactional programming language with implicit transactions, strong isolation, and a multiprocessor implementation.

4.5 Conclusions

The Atomos programming language allows for parallel programming utilizing solely transactional memory without locks. Atomos provides strong atomicity by default, while providing mechanisms to reduce isolation when necessary for features such as loop speculation. Atomos allows programs to specify watch-sets for scalable conditional waiting. The Atomos virtual machine implementation uses violation handlers to recover from expected violations without necessarily rolling back. These rich transactional programming features come with little performance impact compared to the transactional interpretation of Java while giving much more flexibility in creating scalable transactional programs. Chapter 5 will explore how these features can be used to implement the concept of semantic concurrency control on top of the Atomos programming language.

Chapter 5

Semantic Concurrency Control for Transactional Memory

> None of my inventions came by accident. I see a worthwhile need to be met and I make trial after trial until it comes. What it boils down to is one per cent inspiration and ninety-nine per cent perspiration.
>
> – Thomas A. Edison [100]

Although the promise of transactional memory is an easier-to-use programming model, the earlier JavaT and Atomos evaluations have focused on applications where short critical sections have been converted to transactions [30]. For transactional memory to have a real impact, it should not focus on competing with existing hand-tuned applications but emphasize how transactions can make parallel programming easier while maintaining comparable performance [39]. Building efficient parallel programs is difficult because fine-grained locking is required for scaling, which burdens programmers with reasoning about operation interleaving and deadlocks. Large critical regions make it easier on programmers, but degrade performance. However, long-running transactions promise the best of both worlds: a few coarse-grained atomic regions speculatively executing in parallel.

While programming with fewer, longer transactions can make it easier to create correct parallel programs, the downside is that updates to shared state within these transactions can lead to frequent data dependencies between transactions and more lost work when there are conflicts. The dependencies can arise from both the program's own shared data structures as well as underlying library and runtime code. Often the implementation of these underlying structures is opaque to a programmer, so eliminating dependencies is difficult.

As discussed in Chapter 4, open nesting [24, 97] can be used to expose updates to shared data structures early, before commit, and reduce the length of dependencies. This violates the isolation property of transactions and can lead to incorrect and unpredictable programs. However, structured use of open-nested transactions can give the performance benefits of reduced isolation while preserving the semantic benefits of atomicity and serializability for the programmer.

This chapter will discuss how *semantic concurrency control* and *multi-level transactions* can be combined with object-oriented encapsulation to create data structures that maintain the transactional properties of atomicity, isolation, and serializability by changing unneeded memory dependencies into logical dependencies on abstract data types. In addition, at times when full serializability is not required for program correctness, isolation between transactions can be relaxed to improve concurrency. Simple examples like global counters and unique identifier (UID) generators illustrate the usefulness of reduced isolation. The UID example is quite similar to the monotonically increasing identifier problem that the database community uses to demonstrate the trade-offs between isolation and serializability [47].

To illustrate the need for semantic concurrency control when programming with long transactions, a parallelization of a high contention variant of the SPECjbb2000 benchmark is presented [121]. This parallelization includes both the use of `Map` and `SortedMap` as well as the simpler examples of global counters and unique identifier (UID) generation. While the abstract data type examples show how transactional properties can be preserved, the counter and UID examples illustrate how selectively reducing isolation and forgoing serializability can be beneficial as well.

5.1 Supporting Long-Running Transactions

The database community has studied the problem of frequent dependency conflicts within long-running transactions. The database literature presents semantic concurrency control as one solution to the long-running transaction problem. This section describes the evolution of semantic concurrency control, drawing similarities between the problems of databases and the problems of transactional memory. An example will be presented to show how these ideas can be applied directly to transactional memory.

5.1.1 Database Concurrency Control

Isolation, one of the four ACID properties of database transactions, means that changes made by a transaction are not visible to other transactions until that transaction commits. An important derivative property of isolation is serializability, which means that there is a serial ordering of commits that would result in the same final outcome. Serializability is lost if isolation is not preserved, because if a later transaction sees uncommitted results that are then rolled back, the later transaction's outcome depends on data from a transaction that never committed, which means there is no way a serial execution of the two transactions would lead to the same result.

One method for databases to maintain isolation and therefore serializability, is strict two-phase locking. In this form of two-phase locking, the growing phase consists of acquiring locks before data access and the shrinking phase consists of releasing locks at commit time [47].

While simple, this isolation method limits concurrency. Often transactions contain sub-operations, known as nested transactions, which can access the state of their parent transaction without conflict, but which themselves can cause dependencies with other transactions. Moss showed how two-phase locking could be used to build a type of nested transaction where sub-operations could become child transactions, running within the scope of a parent, but able to rollback independently, therefore increasing concurrency (called *closed nesting*) [95]. Gray concurrently introduced a type of nested transaction where the child transaction could commit before the parent, actually reducing isolation and therefore further increasing concurrency because the child could logically commit results based on a parent transaction that could later abort (called *open nesting*) [46].

Open-nested transactions may seem dangerous — exposing writes from a child transaction before the parent commits and discarding any read dependencies created by the child — but they can be very powerful if used correctly. Trager notes how System R used open nesting "informally" by releasing low-level page locks before transactions commit, in violation of strict two-phase locking [124]. System R protected serializability through higher-level locks that are held until commit of the parent transaction, with compensating transactions used to undo the lower-level page operations in case the parent transaction needed to be rolled back.

System R's approach was later formalized as *multi-level transactions*; protecting serializability through locks at different layers [130, 98]. Going a step further and incorporating knowledge about the way specific data structures operate allowed semantically non-conflicting operations to execute concurrently; this was called *semantic currency control* [129, 114]. Finally, *sagas* focused on using compensating transactions to decompose a long-running transaction into a series of smaller, serial transactions [45].

5.1.2 Concurrent Collection Classes

Beyond parallel databases, another area of research in concurrency is data structures. Easier access to multiprocessor systems and programming languages, like Java, that include threads have brought attention to the subject of concurrent collection classes. One major area of effort was `util.concurrent` [81], which became the Java Concurrency Utilities [69]. The original work within `util.concurrent` focused on `ConcurrentHashMap` and `ConcurrentLinkedQueue`, the later based on work by [91]. However, the upcoming JDK 6 release extends this to include a `ConcurrentSkipListMap` that implements the new `NavigableMap` interface that is an extension `SortedMap`.

The idea behind `ConcurrentHashMap` is to reduce contention on a single size field and frequent collisions in buckets. The approach is to partition the table into many independent segments, each with their own size and buckets. This approach of reducing contention through alternative data structure implementations has been explored in the transactional memory community as well as shown below.

A similar partitioning approach is used to implement the `size` method in Click's `NonBlockingHashMap` [32].

5.1.3 Transactional Memory

There has been some work at the intersection of transactional memory and concurrent data structures. Adl-Tabatabai et al. used a `ConcurrentHashMap`-like data structure to evaluate their STM system [2]. Kulkarni et al. suggested the use of open-nested transactions for queue maintenance for Delaunay mesh generation [77]. While this work addressed issues with specific structures, it did not provide a general framework for building transactional data structures.

Pausing transactions was suggested as an alternative to open nesting for reducing isolation between transactions by Zilles and Baugh in [133]. Pausing could be used in the place of open nesting to implement semantic concurrency control, but because pausing does not provide any transactional semantics, traditional methods of moderating concurrent access to shared state, such as lock tables, would need to be used.

Recently, Moss has advocated a different approach less focused on specific data structures. Based on his experience with closed-nested transactions [95], multi-level transactions [98], and transactional memory [61], he has been advocating the use of abstract locks built with open-nested transactions for greater concurrency. This chapter builds on this concept and develops a set of general guidelines and mechanisms for practical semantic concurrency in object-oriented languages. Also included is an evaluation of a full implementation of collection classes for use in SPECjbb2000.

5.1.4 The Need for Semantic Concurrency Control

To understand how ideas from the database community can be applied to transactional memory, consider a hypothetical `HashTable` class:

```
class HashTable {
    Object get  (Object key) {...};
    void   put  (Object key, Object value) {...}; }
```

Semantically speaking, `get` and `put` operations on different keys should not cause data dependencies between two different transactions. Taking advantage of this would be utilizing semantic concurrency control and is based on the fact that such operations are commutative.

The problem is that semantically independent operations may actually be dependent at the memory level due to implementation decisions. For example, hash tables typically maintain a load factor which relies on a count of the current number of entries. If a traditional `java.util.HashMap`-style implementation is used within a transaction, semantically non-conflicting inserts of new keys will cause a memory-level data dependency as both inserts will try and increment the internal size field. Similarly, a `put` operation can conflict with other `get` and `put` operations accessing the same bucket.

Alternative `Map` implementations built especially for concurrent access such as `ConcurrentHashMap`, use multiple hash table segments internally to reduce contention. As mentioned above, others have used similar techniques in transactional contexts to reduce conflicts on a single size field [2]. Unfortunately, while the segmented hash table approach statistically reduces the chances of conflicts in many cases, its does not eliminate them. In fact, the more updates to the hash table, the more segments likely to be touched. If two long-running transactions perform a number of insert or remove operations on different keys, there is a large probability that at least one key from each transaction will end up in the same segment, leading to memory conflicts on the segment's size field.

The solution is to use multi-level transactions. The low-level transactions are open-nested and used to record local changes and acquire higher-level abstract data type locks. The high-level transaction then uses these locks to implement semantic concurrency control.

In the `HashTable` example, the `get` operation takes a read lock on the key and retrieves the appropriate value, if any, all within an open-nested transaction. The `put` operation can use a thread-local variable to store the intent to add a new key-value pair to the table, deferring the actual operation. If the parent transaction eventually commits, a *commit handler* is run that updates the `HashTable` to make its changes visible to other transactions, as well as aborting other transactions that hold conflicting read locks. If the parent transaction is aborted, an *abort handler* rolls back any state changed by open-nested transactions. Note that this approach of optimistic conflict detection with redo logging is not the only option, as will be discussed in Section 5.4.1. Please refer back to Section 4.1.4 for details on the commit and abort handler support in the Atomos programming language.

Before applying multi-level transactions, an unnecessary memory-level conflict would abort the parent transaction. Now, memory-level rollbacks are confined to the short-running, open-nested transaction on `get` and the closed-nested transaction that handles committing `put` operations. In the `get` case, the parent does not rollback, and the `get` operation is simply replayed. In the `put` case, only the commit handler can have memory-level conflicts, and it too can be replayed without rolling back the parent transaction. Note that semantic conflicts are now handled through code in the commit handler that explicitly violates other transactions holding locks on modified keys. The responsibility for isolation, and therefore serializability, has moved from the low-level transactional memory system to a higher-level abstract data type.

To summarize, a general approach to building transactional versions of abstract data types is as follows:

1. take semantic locks on read operations
2. check for semantic conflicts while writing during commit
3. clear semantic locks on abort and commit

Having a general approach is more desirable than relying on data structure specific solutions, like segmented hash tables. For example, the `SortedMap` interface is typically implemented by some variant of a balanced binary tree. Parallelizing a self-balancing tree would involve detailed analysis of the implementation and solving issues like conflicts arising from rotations. Semantic concurrency control avoids these issues by allowing the designer to reuse existing, well-designed and tested implementations.

The following section will discuss more about how to approach semantic concurrency control, covering the semantic operations involved with Java collection classes as well as an implementation of semantic locks.

5.2 Transactional Collection Classes

Simply accessing data structures within a transaction will achieve atomicity, isolation, and serializability, but long-running transactions will be more likely to violate due to the many dependencies created within the data structure. Simply using open nesting to perform data structure updates would increase concurrency, but prevents users from atomically composing multiple updates, as modifications will be visible to other transactions. Users could apply the concepts of semantic concurrency control to their data structures but the improper use of semantic concurrency control can potentially lead to deadlock or livelock issues similar to traditional concurrency control methods.

Fortunately, shared data in parallel programs is usually accessed through abstract data types. By encapsulating the implementation of semantic concurrency control within libraries, the programmer can benefit from well engineered implementations without the need to become an expert in semantic concurrency control implementation. This section presents the design and implementation of such a library based on the abstract data types for object collections provided by Java. These *transactional collection classes* leverage semantic knowledge about abstract data types to allow concurrent and atomic access, without the fear of long-running transactions frequently violating.

Creating a transactional collection class involves first identifying semantic dependencies, namely which operations must be protected from seeing each other's effects. The second step is to enforce these dependencies with semantically meaningful locks. In this section, we discuss these steps in the creation of the `TransactionalMap`, `TransactionalSortedMap`, and `TransactionalQueue` transactional collection classes.

5.2.1 TransactionalMap

The `TransactionalMap` class allows concurrent access to a `Map` from multiple threads while allowing multiple operations from within a single thread to be treated as a single atomic transaction. `TransactionalMap` acts as a wrapper around existing `Map` implementations, allowing the use of special purpose implementations.

Determining Semantic Conflicts

Classes such as `TransactionalMap` can be built by determining which operations cannot be reordered without violating serializability. The first step is to analyze the `Map` abstract data type to understand which operations commute under which conditions. Semantic locks are then used to preserve serializability of non-commutative operations based on these conditions. To understand which `Map` operations can be reordered to build `TransactionalMap`, a multi-step categorization of the operations is performed as described below.

The first categorization of operations is between *primitive* or *derivative* methods. Primitive methods provide the fundamental operations of the data structure while the derivative methods are conveniences built on top of primitive methods. For example, operations such as `isEmpty` and `putAll` can be implemented using `size` and `put`, respectively, and need not be considered further. In the case of `Map`, this categorization helps us reduce the dozens of available methods to those shown in the left column of Table 5.1.

The second categorization is between *read-only* methods and those that `write` logical state of the `Map`. Since read-only operations always commute, the writing methods affect the serializability of each read method, so conflict detection efforts are focused there. In Table 5.1, read and write operations are listed in the left column, showing when they conflict with the write operations in the `put` and `remove` columns.

The `put` and `remove` operations can conflict with methods that read keys, such as `containsKey`, `get`, and `entrySet.iterator.next`. Note that even the non-existence of a key, as determined by `containsKey`, conflicts with the addition of that key via `put`. Similarly, in cases where `put` and `remove` update the semantic size of the `Map`, these methods conflict with operations that reveal the semantic size, namely the `size` and `entrySet.iterator.hasNext`. `entrySet.iterator.hasNext` reveals the size indirectly since it allows someone to count the number of semantic entries in the `Map`. Typically this is used by transactions that enumerate the entire `Map`, which conflict with a transaction that adds or removes keys.

Implementing Semantic Locks

Up to this point, the focus has been on analyzing the behavior of the `Map` abstract data type. Such analysis is largely general and can be used with a variety of implementation strategies. Now the discussion will shift to how this analysis can be used in a specific `TransactionalMap` class implementation. Alternative implementation strategies are discussed in Section 5.4.1.

The discussion in Section 5.1 concluded that dependencies must be released on data structure internals (using open nesting) to avoid unnecessary memory conflicts. To maintain correctness, semantic locks are used to implement multi-level transactions, preserving the logical dependencies of the abstract data type.

In the analysis of `Map`, the ability to reorder method calls depended on two semantic properties: size and the key being operated on. While these choices are `Map` specific, other classes should have similar concepts of abstract state.

Table 5.2 shows the conditions under which locks are taken during different operations. Read operations lock abstract state throughout the transaction. Write operations detect conflicts at commit time by examining the locks held by other transactions. If other transactions have read abstract state being written by the committing transaction, there is a conflict, and the readers are aborted to maintain isolation. For example, a transaction that calls the `size` method acquires the size lock and would conflict with any committing transaction that changes the size (e.g., `put` or `remove`).

Table 5.3 summarizes the internal state used to implement `TransactionalMap`. The `map` field is simply a reference to the wrapped `Map` instance containing the committed state of the map. Any read operations on the `map` field are protected by the appropriate key and size locks. These locks are implemented by the `key2lockers` and `sizeLockers` fields. These fields are shared, so that transactions can detect conflicts with each other, but encapsulated to prevent unstructured access to this potentially isolation-reducing data.

To maintain isolation, the effects of write operations are buffered locally in the current transaction. The `storeBuffer` field records the results of these write operations. Almost all read operations need to consult the `storeBuffer` to ensure they return the correct results with respect to the transaction's previous writes. The one exception is `size`, which instead consults the `delta` field, providing the difference in size represented by the `storeBuffer` operations.

Commit and abort handlers are critical to the correct maintenance of transactional classes. When a transaction is aborted, a compensating transaction must be run to undo changes made by earlier open-nested transactions, in this case releasing semantic locks and clearing any locally buffered state. The `keyLocks` field locally stores locks to avoid explicitly enumerating `key2lockers` when performing this compensation. Commit handlers are used to perform semantic conflict detection, as described above, to release the committing transaction's locks after it has completed and to merge the locally buffered changes into the underlying data structure.

The owner of lock is the top-level transaction at the time of the read operation, not the open-nested transaction that actually performs the read. This is because the open-nested transaction will end soon, but we need to record that the outermost parent transaction needs to be aborted if a conflict is detected. Indeed, it is the handlers of the top-level transaction, whether successful or unsuccessful, that are responsible for releasing any locks taken on its behalf by its children.

One of the most complicated parts of `TransactionalMap` was the implementation of `Iterator` instances for the `entrySet`, `keySet`, and `values`. The iterators need to both enumerate the underlying map with modifications for new or deleted values from the `storeBuffer` and enumerate the `storeBuffer` for newly added keys. The iterator takes key locks as necessary as values are returned by the `next` methods. The iterator also takes the size lock if `hasNext` indicates that the entire `Set` was enumerated.

5.2.2 TransactionalSortedMap

The `TransactionalSortedMap` class extends `TransactionalMap` to provide concurrent atomic access by multiple non-conflicting readers and writers to implementations of the Java `SortedMap` interface. The `SortedMap` abstract data type extends `Map` by adding support for ordered iteration, minimum and maximum keys, and range-based sub-map views.

Determining Semantic Conflicts

The `SortedMap` interface extends the `Map` interface by adding new methods for dealing with sorted keys, and also by defining the semantics of existing methods such as `entrySet` to provide ordering. Mutable `SortedMap` views returned by `subMap`, `headMap`, and `tailMap` also have to be considered in the analysis. In Table 5.4, a similar categorization is performed of abstract data type operations as in the last section, focusing on the new primitive operations and on the operations with changed behavior such as `entrySet`. New operations that are derivative, such as `firstKey`, are omitted.

The categorization shows that all of the new operations are read-only. In addition to the key and size properties of `Map`, methods now also read ranges of keys as well as noting the first and last key of the `SortedMap`. The existing write operations `put` and `remove` are now updated to show their effects on ranges of keys as well as the endpoint keys. Specifically, a `put` or `remove` operation conflicts with any operation that reads a range of keys that includes the key argument of the `put` or `remove`. It is important to note that ranges are more that just a series of keys. For example, inserting a new key in one transaction that is within a range of keys iterated by another transaction would violate serializability if the conflict was not detected. In addition to the range

Write / *Read*	put	remove
containsKey	if put adds a new entry with same key	if remove takes away entry with same key
get	if put adds a new entry with same key	if remove takes away entry with same key
size	if put adds a new entry	if remove takes away an entry
entrySet.iterator.hasNext	if hasNext is false and put adds a new entry	remove takes away key in iterated range
entrySet.iterator.next	put adds key in iterated range	remove takes away key in iterated range
Write		
put	if both write to the same key	if both operate on the same key
remove	if both operate on the same key	if both remove the same key

Table 5.1: Semantic operational analysis of the `Map` interface showing the conditions under which conflicts arise between primitive operations. Both read and write operations are listed along the left side but only write operations are listed across the top. The read operations are omitted along the top since read operations do not conflict with other read operations. If the condition is met, there needs to be an ordering dependency between the two operations. For example, the upper left condition says that if a `put` operation adds an entry with a new key in one transaction and another transaction calls `containsKey` on that same key returning false, there is a conflict between the transactions because they are not serializable if the `put` operations commits before the `containsKey` operation, which would be required to return true in a serializable schedule.

Methods	**Read Lock**	**Write Conflict**
Read		
containsKey	key lock on argument	
get	key lock on argument	
size	size lock	
entrySet.iterator.hasNext	size lock on false return value	
entrySet.iterator.next	key lock on return value	
Write		
put	key lock on argument	key conflict based on argument size conflict if size decreases
remove	key lock on argument	key conflict based on argument size conflict if size decreases

Table 5.2: Semantic locks for `Map` describe read locks that are taken when executing operations as well as lock based conflict detection that is done by writes at commit time. For example, the `containsKey`, `get`, `put`, and `remove` operations take a lock for the key that was passed as an argument to these methods. When a transaction containing `put` or `remove` operations commits, it aborts other transactions that hold locks on the keys it is adding or removing from the `Map`, as well as on other transactions that have read the size of the `Map` if it is growing or shrinking.

Category	**Field**	**Description**
Committed State		*committed state visible to all transactions*
	`Map map`	the underlying `Map` instance
Shared Transaction State		*state managed by open nesting, encapsulated in* ***`TransactionalMap`***
	`Map key2lockers`	map from keys to set of lockers
	`Set sizeLockers`	set of size lockers
Local Transaction State		*state visible by the local thread*
	`Set keyLocks`	set of key locks held by the thread
	`Map storeBuffer`	map of keys to new values, special value for removed keys
	`int delta`	change in size due to changes in `storeBuffer`

Table 5.3: Summary of `TransactionalMap` state.

keys, `put` and `remove` can affect the first and last key by respectively adding or removing new minimum or maximum elements, thus conflicting with operations that either explicitly request these values with `firstKey` or `lastKey` or implicitly read these values through an iterator, including iterators of views.

Implementing Semantic Locks

Table 5.5 shows the conditions under which locks are taken during different `SortedMap` operations. In addition to the key and size locks of `Map`, semantic locks have been added for the first and last key as well as for key ranges.

Table 5.6 summarizes the extensions to the internal state of `TransactionalMap` used to implement `TransactionalSortedMap`. The `sortedMap` field is a `SortedMap`-typed version of the `map` field from `TransactionalMap`. The `comparator` field is used to compare keys either using the `Comparator` of the underlying `SortedMap` if one is provided, or using `Comparable` if the `SortedMap` did not specify a `Comparator`. Note that the `comparator` is established during construction and thereafter is read-only so no locks are required to protect its access.

The key-based locking of `TransactionalMap` is extended in `TransactionalSortedMap` by key range locking, provided by the `rangeLockers` field. As with key locks, writers must determine which subset of outstanding transactions conflict with their updates. In the implementation evaluated in this chapter, a simple `Set` was chosen to store the range locks, meaning updates to a key must enumerate the set to find matching ranges for conflicts. An alternative would have been to use an interval tree to store the range locks, but the extra complexity and potential overhead seemed unnecessary for the common case. The endpoint-based locking of `TransactionalSortedMap` is provided by the `firstLockers` and `lastLockers` fields. Like size locking, and unlike key range locking, endpoint locking does not not require any search for conflicting transactions, since endpoint lockers are conflicting whenever the corresponding endpoint changes.

The local transaction state for `TransactionalSortedMap` consists primarily of the `rangeLocks` field which allows efficient enumeration of range locks for cleanup on commit or abort, without enumerating the potentially larger global `rangeLockers` field. In addition, the `sortedStoreBuffer` provides a `SortedMap` reference to the `storeBuffer` field from `TransactionalMap` in order to provide ordered enumeration of local changes.

As with `TransactionalMap`, one of the more difficult parts of implementing `TransactionalSortedMap` was providing iteration. In order to provide proper ordering, iterators must simultaneously iterate through both the `sortedStoreBuffer` and the underlying `sortedMap`, while respecting ranges specified by views such as `subMap`, and take endpoint locks as necessary.

5.2.3 TransactionalQueue

In the database world, SQL programs can request reduced isolation levels in order to gain more performance. Similarly, sometimes in transactional memory it is useful to selectively reduce isolation. One example is in creating a `TransactionalQueue`. The idea is inspired by a Delaunay mesh refinement application that takes work from a queue and may add new items during processing. Open-nested transactions could be used to avoid conflicts by immediately removing and adding work to the queue [77]. However, if transactions abort, the new work added to the queue is invalid but may be impossible to recover since another transaction may have dequeued it. The `TransactionalQueue` provides the necessary functionality by wrapping a `Queue` implementation with a `Channel` interface from the `util.concurrent` package [81].

Providing the simpler `Channel` interface lowers the design complexity by eliminating unnecessary `Queue` operations that do not make sense for a concurrent work queue, such as random access operations, instead only providing operations to enqueue and dequeue elements. To improve concurrency, strict ordering is not maintained on the queue, resulting in fewer semantic conflicts between transactions. As Table 5.7 shows, if transactions confine themselves to the common `put` and `take` operations, no semantic conflicts can ever occur. The only semantic conflict detected is when one transaction detects an empty `Queue` via a `null` result from `peek` or `poll`, and another transaction adds a new element with `put` or `offer`.

Table 5.9 summarizes the internal state used to implement `TransactionalQueue`.

Write *Read*	put	remove
entrySet.iterator.hasNext	hasNext is false and put adds new lastKey	hasNext returns true about lastKey and remove takes away lastKey
entrySet.iterator.next	put adds key in iterated range	remove takes away key in iterated range
comparator		
subMap.iterator.next	put adds key in iterated range	remove takes away key in iterated range
headMap.iterator.next	put adds key in iterated range	remove takes away key in iterated range
tailMap.iterator.next	put adds key in iterated range	remove takes away key in iterated range
tailMap.iterator.hasNext	hasNext is false and put adds new lastKey	hasNext returns true about lastKey and remove takes away lastKey
lastKey	put adds a new lastKey	remove takes away the lastKey

Table 5.4: Semantic operational analysis of the `SortedMap` interface. This focuses on new and changed primitive operations relative to the `Map` interface in Table 5.1.

Methods	**Read Lock**	**Write Conflict**
Read Only		
entrySet.iterator.hasNext	last lock on false return value	
entrySet.iterator.next	range lock over iterated values, first lock	
comparator		
subMap.iterator.next	range lock over iterated values	
headMap.iterator.next	range lock over iterated values, first lock	
tailMap.iterator.next	range lock over iterated values	
tailMap.iterator.hasNext	last lock on false return value	
firstKey	first lock	
lastKey	last lock	
Write		
put	key lock on argument	key and range conflicts on argument first and last lock on endpoint change size conflict on increases
remove	key lock on argument	key and range conflicts on argument first and last lock on endpoint change size conflict on decreases

Table 5.5: Semantic locks for `SortedMap`. This focuses on new and changed primitive operations relative to the `Map` interface in Table 5.2.

Category	**Field**	**Description**
Committed Transactional State		*committed state visible to all transactions*
	`SortedMap sortedMap`	the underlying `SortedMap` instance
	`Comparator comparator`	read-only `Comparator` instance
Shared Transactional State		*state managed by open nesting encapsulated in* ***`TransactionalMap`***
	`Set firstLockers`	set of first key lockers
	`Set lastLockers`	set of last key lockers
	`Set rangeLockers`	set of last key lockers
Local Transactional State		*state visible by the local thread*
	`Set rangeLocks`	set of range locks held by the thread
	`SortedMap sortedStoreBuffer`	sorted map of keys to new values, special value for removed keys

Table 5.6: Summary of `TransactionalSortedMap` state. This focuses on the additions of state of the `TransactionalMap` superclass in Table 5.3.

Read \ *Write*	put	take	poll
peek	if peek returned null		
Write			
put			
take			
poll	if poll returned null		

Table 5.7: Semantic operational analysis of the `Channel` interface showing the conditions under which conflicts arise with write operations listed across the top.

Methods	**Read Lock**	**Write Conflict**
Read		
peek	if empty	
Write		
put		if now non-empty
take		
poll	if empty	

Table 5.8: Semantic locks for `Channel` describe empty locks that are taken when executing operation, as well as lock based conflict detection that is done by writes at commit time.

The `queue` field holds the current committed state of an underlying `Queue` instance. The `emptyLockers` field tracks which transactions have noticed when the queue is empty. The `addBuffer` field tracks new items that need to be added to the queue when the parent transaction commits, while the `removeBuffer` tracks removed items that should be returned to the queue of the parent transaction aborts. While simple in construction compared to the fully serializable `TransactionalMap` and `TransactionalSortedMap` classes, the Delaunay example shows the benefits of having a transactional aware queue that allows multiple operations within a single transaction.

Category	**Field**	**Description**
Committed State		*committed state visible to all transactions*
	`Queue queue`	the underlying `Queue` instance
Shared Transaction State		*state managed by open nesting encapsulated in `TransactionalQueue`*
	`Set emptyLockers`	set of empty lockers
Local Transaction State		*state visible by the local thread*
	`List addBuffer`	list of locally added elements
	`List removeBuffer`	list of locally removed elements

Table 5.9: Summary of `TransactionalQueue` state.

5.3 Semantics for Transactional Collection Classes

This section discusses the functionality necessary to implement transactional collection classes. These mechanisms and their use are referred to as *transactional semantics*. While all transactional memory systems offer an interface for denoting transactional regions, some of them already provide elements of the identifying transactional semantics.

5.3.1 Nested Transactions: Open and Closed

Some systems implement *flat* closed-nested transactions: the child simply runs as part of the parent without support for partial rollback. However, to reduce lost work due to unnecessary conflicts, the implementation from this chapter needs partial rollback of the commit handlers run as closed-nested transactions. That way, any conflicts during update of the underlying data structure will only roll back the commit handler and not the entire parent.

Open nesting is probably the most significant building block for semantic concurrency control. It is the enabling feature that allows transactions to create semantic locks without retaining memory dependencies that will lead to unnecessary conflicts. However, while open-nested transactions are a necessary feature for supporting semantic concurrency control, they are not sufficient without some way of cleaning up semantic locks when the overall transaction finishes — this is the purpose of commit and abort handlers.

For more information on closed nesting and open nesting in Atomos, please see Section 4.1.1 and Section 4.1.3, respectively.

5.3.2 Commit and Abort Handlers

Commit and abort handlers allow code to run on the event of successful or unsuccessful transaction outcome, respectively. Transactional collection classes use these handlers to perform the required semantic operations for commit and abort, typically writing the new state on commit, performing compensation on abort, and releasing semantic locks in both cases.

Commit handlers typically run in a closed-nested transaction, so that any memory conflicts detected during their updates to global state do not cause the re-execution of the parent. Handlers execute at the end of the parent transaction so it has visibility into the parent's state. This is useful for cleaning up any thread-local values.

Abort handlers typically run in an open-nested transaction. As with commit handlers, they are nested within the parent so they can access the parent's state before it is rolled back. Open nesting allows the abort handler to undo any changes performed by the parent's open-nested transactions; otherwise, any work done by the abort handler would simply be rolled back along with the parent.

When a commit or abort handler is registered, it is associated with the current level of nesting. If the nested transaction is aborted, the handlers are simply discarded without executing — rollback should clear the state associated with the handlers. If the nested transaction commits, the handlers are associated with the parent so necessary updates/compensation will happen when the parent completes/aborts.

Discarding newly registered handlers prevents a handler from running in unexpected situations. Since a conflict could be detected at any time, an abort handler could be invoked at any point in the execution of a transaction. Conceptually, this is very similar to the problem of reentrancy of Unix signal handlers; it is difficult to insure that data structure invariants hold. With signal handlers, the approach is usually to do very little within handlers except to note that the signal happened, letting the main flow of control in the program address the issue. Fortunately, nested transactions and encapsulation can provide more guarantees about the state of objects. If the only updates to the encapsulated state, such as the local tables and store buffers, are made with open-nested transactions, then when an abort handler runs it can be sure that these encapsulated data structures are in a known state.

Discarding newly registered handlers on abort interacts with using abort handlers for compensation of non-transactional operations. However, these operations should not have been performed during the body of the transaction but rather during commit handlers. While logically this makes sense for output operations deferred to the end of the transaction, it also works for input operations as they can be performed in the commit handler of an open-nested transaction that registers an abort handler to push back input as needed. While handlers are not a general solution for handling all cases of non-transactional operations, these semantics cover two frequently cited examples of using handlers to mix I/O and transactions.

Some systems use two-phase commit as part of executing commit handlers. Two-phase commit breaks a transaction commit into two parts: *validation* and *commit*. After validation is completed, the transaction is assured that it will be able to commit. Typically, commit handlers are run in the commit phase after validation. This guarantees that any non-transactional action, such as I/O, do not need to worry that the parent will get violated after an irreversible action is performed. Note that for building transactional collection classes with semantic concurrency control, there is no need to perform any non-transactional operations, only updates to data structures in memory, so that two phase commit is not strictly required, although its presence is not a problem.

For more information on Atomos commit and abort handlers, please see Section 4.1.4

5.3.3 Program-directed Transaction Abort

Transactional memory systems can automatically abort transactions with serializability conflicts. Some systems provide an interface for transactions to abort themselves, perhaps if they detect a problem with the consistency property of ACID transactions. Semantic concurrency control requires the additional ability for one transaction to abort another when semantic-level transactional conflicts are detected. Specifically for the implementation discussed above, an open-nested transaction needs a way to request a reference to its top-level transaction that can be stored as the owner of a lock. Later, if another transaction detects a conflict with that lock, the transaction reference can be used to abort the conflicting transaction.

For more information on how Atomos supports program-directed transaction abort, see the `Transaction` class and its `getCurrentTransaction` and `abort` methods in Section 4.1.5.

5.4 Serializability Guidelines

The most difficult part of semantic concurrency control is analyzing the abstract data type to determine the rules for commutative operations and determining a set of semantic locks to properly preserve commutativity. However, once this is done, the actual implementation of semantic concurrency control via multi-level transactions is fairly straightforward using a simple set of rules:

- The underlying state of the data structure should only be read within an open-nested transaction that also takes the appropriate semantic locks. This ensures that the parent transaction contains no read state on the underlying state that could cause memory-level conflicts.

- The underlying state of the data structure should only be written by a closed-nested transaction in a commit handler. This preserves isolation since semantic changes are only made globally visible when the parent transaction commits.

- Because write operations should not modify the underlying data structure, write operations need to store their state in a transaction-local buffer. If semantic locks are necessary because the write operation logically includes a read operation as well, the locks should be taken in an open-nested transaction, as above.

- The abort handler should clear any changes made with open-nested transactions including releasing semantic locks and clearing any thread-local buffers. Only one abort handler is necessary and it should be registered by the first open-nested transaction to commit.
- The commit handler should apply the buffered changes to the underlying data structure. As it applies the changes, it should check for conflicting semantic locks for the operations it is performing. After it has applied the changes, it follows the behavior of the abort handler, ensuring that the buffer is cleared and that semantic locks are released. As with the abort handler, only a single commit handler is needed, registered on the first write operation.

Note that if reduced isolation is desired, the second rule is typically broken by allowing writes to the underlying state from within open-nested transactions. For example, in the `TransactionalQueue` implementation, the `take` method removed objects from the underlying queue without acquiring a lock.

5.4.1 Discussion

Alternatives to Optimistic Concurrency Control

Detecting conflicting changes at commit time is known as *optimistic concurrency control*. Another approach is to detect conflicts as soon as possible (*pessimistic concurrency control*). In the implementation described above, write operations could detect conflicting semantic locks when the operation is first performed, instead of waiting until commit. A contention management policy can then be used to decide how to proceed. One approach is to have the conflicting write operation wait for the other transaction to complete. However, this leads to the usual problems with locks, such as deadlock. The downside to optimistic concurrency control is that it can suffer from livelock since long-running transactions may be continuously rolled back by shorter ones. Here again, contention management policies can be applied to give repeatedly violated transactions priority. A discussion of contention management in transactional memory systems can be found in [49]. The choice of optimistic concurrency control for semantic-level transactions is independent of the underlying concurrency control in the transactional memory system.

Redo versus undo logging

The approach of buffering changes and replaying them at commit time is a form of *redo logging*, so called because the system repeats the work of the operations in the local buffer on the global state. The alternative is *undo logging*, where the system updates global state in place. If there are no conflicts, the undo log is simply dropped at commit time. If there is a conflict and the transaction needs to abort, the undo log can be used to perform the compensating actions to roll back changes made to the global state by the aborting transaction. In the implementation described above, redo logging was chosen because it is a better fit to optimistic concurrency control, since undo logging requires early conflict detection since only one writer can be allowed to update a piece of semantic state in place at a time. Note that the choice of redo versus undo logging for semantic-level transactions is independent of the underlying logging used by the transactional memory system.

Single versus multiple handlers

The implementation discussed above uses one commit and one abort handler per parent transaction. These handlers know how to walk the underlying lock and buffer structures to perform the necessary work on behalf of all previous operations to the data structure. An alternative is for each operation to provide its own independent handlers.

Moss extends this alternative by proposing that each abort handler should run in the memory context that was present at the end of the child transaction in which it was registered, interleaved with the undoing of the intervening memory transactions [97].

For example, suppose there is a series of operations that make up a parent transaction $AXBYC$, with A, B, and C being memory operations and X and Y being open-nested transactions. Moss suggests that to abort at the end of C, the logically inverse actions $C^{-1}Y^{-1}B^{-1}X^{-1}A^{-1}$ should be performed. Logically, this consists of rolling back the memory operations of C, followed by running the abort handler for Y, followed by rolling back B, then running the abort handler for X, and finally rolling back the memory operations of A.

This extra complexity is unnecessary for the above implementation of semantic concurrency control. Moss's semantics aim to guarantee that the handler will always run in a well-defined context known to the handler at its point of registration. However, the guidelines given provide similar guarantees to handlers since object-oriented encapsulation ensures that only the transaction that registered the handler can make updates to the state that the handler will later access on abort.

Alternative semantic locks

In the initial categorization of `Map` methods into primitive and derivative operations, several methods were considered primitive that semantically speaking are strictly derivative if one focuses on correctness and not performance. After all, it is possible to build a writable `Map` by subclassing `AbstractMap` and implementing only the `entrySet` and `put` methods, although such an association list style implementation would need to iterate the `entrySet` on `get` operations, taking locks on many keys unnecessarily, compromising concurrency while maintaining correctness.

While it might seem to be contradicting the stated methodology, in fact in one considers the expected running time of operations to be part of the semantics of an abstract data type, treating certain methods as primitive is consistent. Certainly if one is happy to have a linear cost for `Map` retrievals, then it is fine to treat `get` as a derived operations. However, if one expects to have close to constant cost for a `Map` and logarithmic for a `SortedMap`, one needs to follow the more typical approach of `HashMap` and `TreeMap` and consider other methods such as `containsKey`, `get`, `size`, and `remove` as primitive operations, which avoids the need to iterate the `entrySet` for these operations.

While the semantic locks derived from the primitive methods in Table 5.1 and Table 5.4 preserve isolation and serializability, they still do not allow the optimal concurrency possible. One limitation is making `isEmpty` a derivative method based on `size`, resulting in `isEmpty` taking a size lock. To see why this is a problem, consider two transactions running this code:

```
    if (!map.isEmpty()) map.put(key, value);
```

These transactions should commute as long as they add different keys, but the current implementation will cause one to abort because of the size lock. However, taking the size lock is necessary for the similar case of two transactions running:

```
    if (map.isEmpty()) map.put(key, value);
```

These `put` operations should not commute because a serial ordering would require that only one would find an empty map. The solution is to make `isEmpty` a primitive operation with its own separate semantic lock that is violated only when the size changes to or from zero.

Note that while the above discussion focuses on the third categorization for the primitive methods, logically this categorization is important for the derivative methods as well. This can expose unexpected concurrency limitations. For example, a straightforward implementation of `entrySet.remove` might use `size` to determine if the `remove` operation actually found a matching key, which is used to calculate the Boolean return value. This would add an unnecessary dependency on the size, which could cause unnecessary conflicts if others concurrently added or removed other keys.

Extensions to `java.util.Map`

C++ programmers often use idioms like

```
    if (map.size()) ...
```

instead of

```
    if (!map.empty()) ...
```

arguably because it is easy for a reader to miss the negation when reviewing code. Similarly, Java programmers frequently use `if (map.size() == 0) ...` instead of `isEmpty`. However, as noted in the previous discussion, using `size` instead of `isEmpty` unnecessarily restricts concurrency.

Similar problems exist for `Map` methods that reveal more information than strictly necessary. For example, since the write operations `put` and `remove` return the old value for a key, they effectively read the key as well. If this return value is unused, this is an unnecessary limitation of semantic concurrency. To be specific, there is no reason that two transactions that write to the same key need to be ordered in any way. For example, it is perfectly acceptable for two transactions to do this:

```
map.put("LastModified", new Date());
```

Logically, these transactions can commit in any order so long as they do not read the "LastModified" key. However, the fact that the old value for a key is returned causes a dependency on the key, causing the two `put` operations to conflict which each other.

The solution is to offer alternative variants to methods such as `put` and `remove` that do not reveal unnecessary information through unused return values, allowing the caller to decide which is appropriate.

TransactionalSet and TransactionalSortedSet

`TransactionalSet` and `TransactionalSortedSet` classes are not discussed at length because they can be built as simple wrappers around the `TransactionalMap` and `TransactionalSortedMap`, respectively, as has been done similarly for `ConcurrentHashSet` implementations built on top of `ConcurrentHashMap`, and even `HashSet` implementations around `HashMap` as found in [44].

Leaking uncommitted data

While the guidelines prevent leaking of uncommitted data between transactions using the same transactional collection class, values used within the class's semantic locks, such as keys or range endpoints, can be visible to the other open-nested transactions operating on the instance. For example, if a newly allocated string is used as a key name in a `TransactionalMap`, the `key2lockers` table would have an entry pointing to an object that is only initialized within the adding transaction. However, if another transaction adds another key that hashes to the same bucket, the table will call `Object.equals` to compare the new key to the existing key, which is uninitialized from this second transaction's point of view.

In [97], Moss proposes making object allocation and type initialization an open-nested transaction, so at least access to this uncommitted object will not violate Java type safety. However, the constructor part of object allocation cannot be safely made part of an open-nested transaction because it could perform arbitrary operations that might require compensation. Moss notes that for some common key classes such as `java.lang.String`, it is safe and even desirable to run the constructor as part of an open-nested allocation, but this is not a general solution.

An alternative is to not directly insert such potentially uncommitted objects into semantic locking tables but instead insert copies. One approach would be to use existing mechanisms such as `Object.clone` or `Serializable` to make a copy, similar to what is proposed by Harris in [54], which uses `Serializable` to copy selected state out of transactions that are about to be aborted. Alternatively, a new interface could be used to request a committed key from an object, allowing it to make a selective copy of a subset of identifying state, rather than the whole object like `clone` or `Serializable`, perhaps simply returning an already committed key.

5.5 Evaluation

This section will use variants of a common transactional memory micro-benchmark, as well as a custom version of SPECjbb2000, designed to have higher contention, to evaluate the performance of transactional collection classes created with semantic concurrency control. High-contention versions of SPECjbb2000 have previously been used for similar purposes [28, 16]. This evaluation includes results from lock-based Java execution for comparison. The results focus on benchmark execution time, skipping virtual machine startup. The single-processor Java version is used as the baseline for calculating speedup.

While the experiments are performed with a specific language, virtual machine, and HTM implementation, the observations and conclusions apply to other hardware transactional memory systems as well.

5.5.1 Map and SortedMap Benchmarks

TestMap is a micro-benchmark based on a description in [2] that performs multi-threaded access to a single `Map` instance. Threads perform a mixture of operations with a breakdown of 80% lookups, 10% insertions, and 10% removals. To emulate access to the `Map` from within long-running transactions, each operation is surrounded by computation. There should be little semantic contention in this benchmark but frequent memory contention within the `Map` implementation, such as the internal size field.

Figure 5.1 summarizes the results for TestMap. As expected, Java with `HashMap` shows near linear scalability because the lock is only held for a small time relative to the surrounding computation. The Atomos `HashMap` result shows what happens when multiple threads try to simultaneously access the `Map` instance, with scalability limited as the number of processors increases because of contention on the `HashMap` size field. Atomos results with a `ConcurrentHashMap` show how some, but not all, scalability can be regained by using the probabilistic approach of a segmented hash table. The Atomos results with a `TransactionalMap` wrapped around the `HashMap` show how the full scalability of the Java version can be regained when unnecessary memory conflicts on the size field are eliminated.

TestSortedMap is a variant of TestMap that replaces lookup operations using `Map.get` with a range lookup using `SortedMap.subMap`, taking the median key from the returned range. As with TestMap, there is little semantic contention as the ranges are relatively small and serve just to ensure there are not excessive overheads from the range locking implementation.

Figure 5.2 shows that Java with a `SortedMap` scales linearly as expected. Atomos with a plain `TreeMap` fails to scale because of non-semantic conflicts due to internal

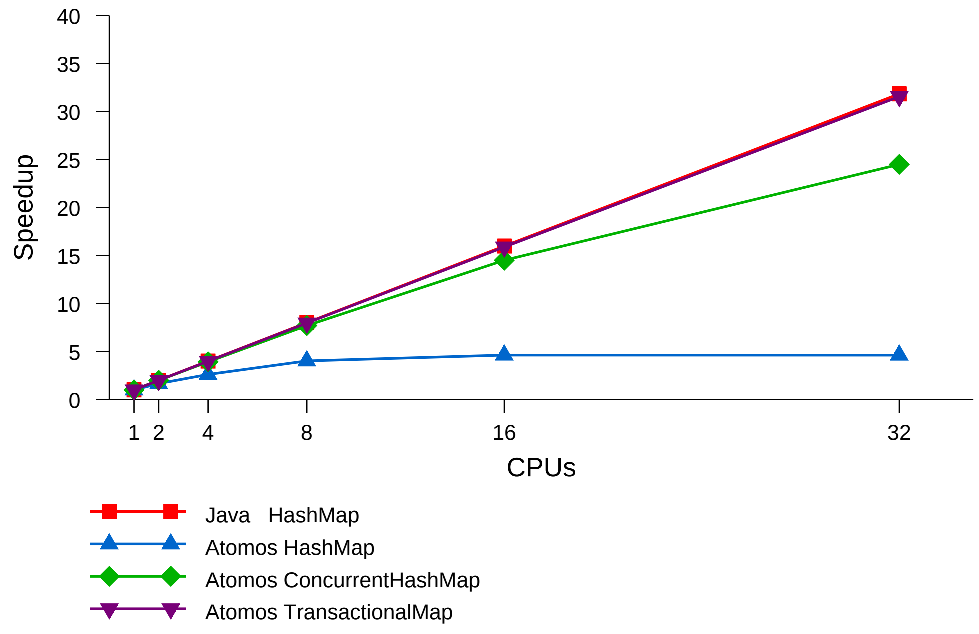


Figure 5.1:
TestMap results show that Atomos can achieve the scalability of Java when the concurrently accessed `HashMap` is wrapped in a `TransactionalMap`. The results also show that using a `ConcurrentHashMap` to probabilistically reduce data dependencies within transactions falls short of the speedup of the semantic concurrency control method of `TransactionalMap`.

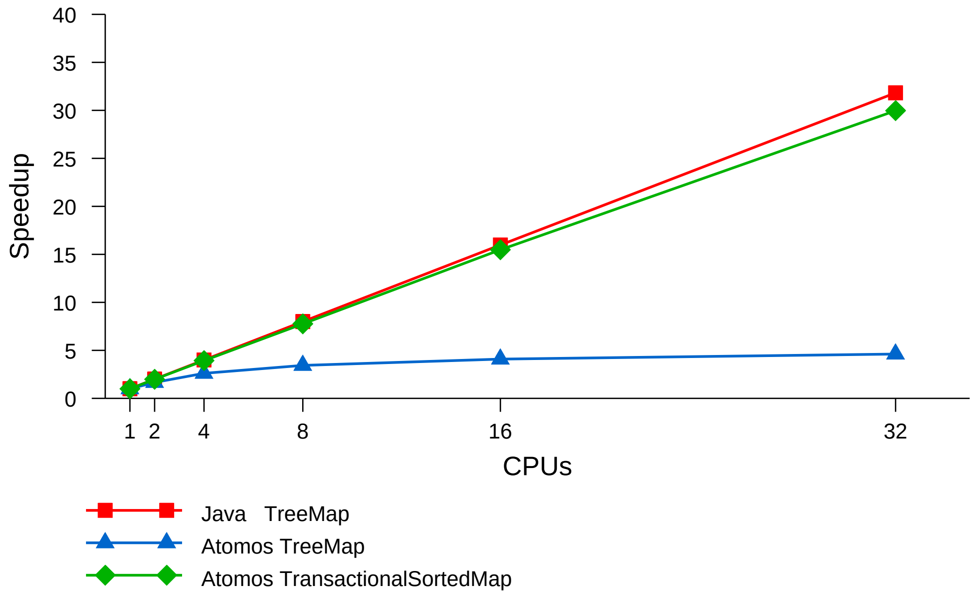


Figure 5.2: TestSortedMap results parallel TestMap showing that `Transactional-SortedMap` provides similar benefits to a concurrently accessed `TreeMap`.

operations such as red-black tree balancing. Finally, Atomos with a `Transaction-alSortedMap` wrapped around a `TreeMap` instance regains the scalability of the Java version.

TestCompound is a variant of TestMap that composes two operations separated by some computation. The results are shown in Figure 5.3. In the Java version, a coarse-grained lock is used to ensure that two operations act as a single compound operation. For Atomos, the entire loop body, including other computation before and after the compound operation, is performed as a single transaction. In this case, the Java version scales poorly since a single lock is held during the computation between the two operations, with little difference to the Atomos `HashMap` result. However, the `TransactionalMap` result shows that transactional collection classes can provide both composable operations and concurrency.

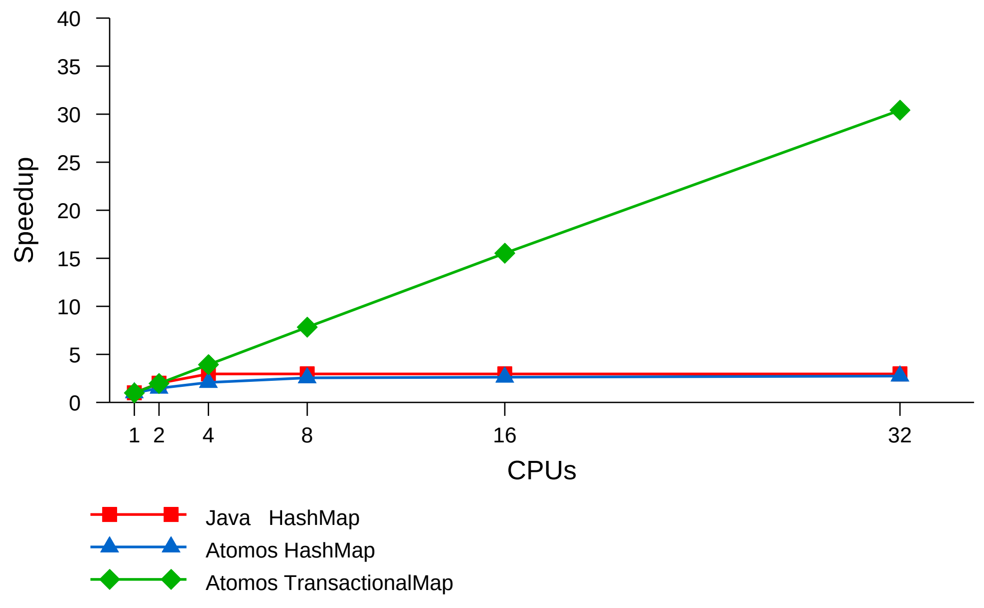


Figure 5.3: TestCompound results shows that Java scalability is limited by use of a coarse-grained lock to protect a compound operation which scales as a single Atomos transaction.

5.5.2 High-contention SPECjbb2000

As discussed before in Section 3.3.5, the SPECjbb2000 benchmark [121] is an embarrassingly parallel Java program with little inherent contention. As a benchmark its focus is to ensure that the underlying virtual machine and supporting system software are scalable. There are only a few shared fields and data structures in SPECjbb2000, each protected by `synchronized` critical regions in the Java version. Each application thread is assigned an independent warehouse to service TPC-C style requests, with only a 1% chance of contention from inter-warehouse requests. The previous results for JavaT in Section 3.3.5 and for Atomos in Section 4.3.1 have shown that a transactional version of SPECjbb2000 can scale as well as Java.

Instead of reusing this embarrassingly parallel version of SPECjbb2000 that largely partitions threads into separate warehouses, this section will use a high-contention version of SPECjbb2000 that has a single warehouse for all operations. In addition, in order to approximate the parallelization of a sequential program, the Atomos version will remove all Java `synchronized` critical regions and instead turn each of five TPC-C operations into atomic transactions. The correctness of this parallelization is easy to reason about even for a novice parallel programmer, because all the parallel code excluding the thread startup and loop setup is now executed within transactions. Both the Java and Atomos versions use `java.util` collection classes in place of the original tree implementation, following the pattern of SPECjbb2005.

Figure 5.4 shows the results for the modified version of SPECjbb2000. First, note that the modifications to use a single warehouse significantly impact the scalability of the Java version, which usually would achieve nearly linear speedup on 32 processors as seen in Section 3.3.5.

The *Atomos Baseline* version is the first cut parallelization with each operation in one long transaction. The performance of this version is limited by a large number of violations from a variety of conflicting memory operations. Techniques described in [25] were used to identify several global counters such as the `District.nextOrder` ID generator as the main sources of lost work due to conflicts.

The *Atomos Open* version addresses the ID generator issue that was the main source of performance degradation in the *Atomos Baseline* version. By wrapping

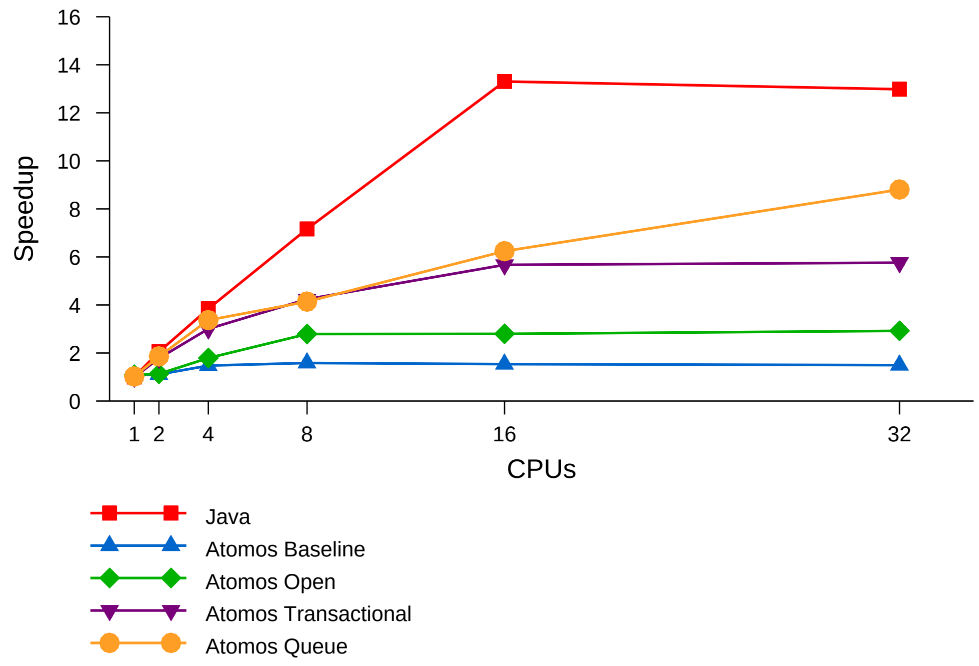


Figure 5.4: SPECjbb2000 results in a high-contention configuration caused by sharing a single warehouse.

SPECjbb2000 Version	**Speedup on 32 CPUs**	**Coding Effort**
Java	13.0	272 synchronized statements
Atomos Baseline	1.5	1 atomic statement
Atomos Open	2.9	4 open statements
Atomos Transactional	5.8	2 TransactionalMap 1 TransactionalSortedMap 2 transactional counters
Atomos Queue	8.8	Change TransactionalSortedMap to a TransactionalMap and a TransactionalQueue (2 new calls: Queue.add & remove)
Atomos Short	25.5	272 atomic statements

Table 5.10: Comparison of coding effort and 32 CPU speedup of various versions of a high-contention configuration of SPECjbb2000. Note that the Atomos Queue parallelization with using long transaction required 14 changes to get a final speedup of 7.3, where the Java version required 272 `synchronized` statements to get a speedup of 11.7.

reads and writes to the these counters in open-nested transactions, the counter semantics were preserved while reducing lost work. This technique was previously discussed in Section 4.1.3. After this change, additional conflict analysis identified three shared `Map` instances that were frequent sources of conflicts: `Warehouse.historyTable`, `District.orderTable`, and `District.newOrderTable`. Also, two "year-to-date" payment counters were found in `Warehouse.ytd` and `District.ytd`. Unlike the unique ID generators, access to these payment fields cannot simply be wrapped in open-nested transactions, since they need to be correctly maintained in the face of the rollback of parent transactions for payment balances to be correctly maintained.

The *Atomos Transactional* version addresses both the `Map` and payment issues using semantic concurrency control. The three mentioned `Map` instances were wrapped with `TransactionalMap` and `TransactionalSortedMap` as appropriate. The `float` payment counter fields were replaced with `TransactionalFloatCounter` objects that follow a similar, though much simpler, design and implementation of the transactional collection classes. The basic idea of the `TransactionalFloatCounter` is that it defers increments to a shared counter until a commit handler at the end of the transaction, while still letting the transaction see its own local changes during the transaction. With these changes, conflict analysis shows that memory-level conflicts are not the primary factor limiting performance, although there is now a semantic conflict that is the bottleneck. Specifically, the semantic conflict is that multiple threads attempt to concurrently remove the first key of the `District.newOrderTable`, and logically only one can succeed, since the first key changes whenever one of them completes.

This semantic dependency occurs because SPECjbb2000 uses a `SortedMap` for the `District.newOrderTable` for two independent purposes. The first is to look up `NewOrder` objects by ID. The second is to age out older `NewOrder` to maintain a constant memory footprint. Fortunately, semantic dependencies, like the memory dependencies, point directly to the source of the performance problem.

The *Atomos Queue* version shows the results of breaking this particular semantic dependency by changing the single `District.newOrderTable` into two data structures, including one with reduced isolation. The first data structure is simply the existing `District.newOrderTable`, but changed to be a `Map` instead of a `SortedMap`. The second data structure is a new `TransactionalQueue` which is used to track an approximate FIFO queue of `NewOrder` objects for aging. The key difference is that the queue ordering is only approximate, which works well in this application, since it really only needs to remove an old `NewOrder` object, not necessarily the oldest.

The use of simple open-nested counters and transactional classes yielded a reasonable speedup for little effort on the high-contention SPECjbb2000 benchmark. Table 5.10 summarizes the effort required for each parallelization attempted. Starting from source code with no concurrency control primitives, the final *Atomos Queue* with long transactions has a speedup of 8.8 on 32 CPUs relative to the single CPU Java version of the program. While the Java version achieves a speedup of 13.0 on 32 CPUs, the programmer has added 272 `synchronized` statements to achieve these results.

A final analysis of the long transaction version of the high-contention SPECjbb-2000 application reveals more opportunities for improvement, such as splitting the TPC-C operations into multiple transactions. Indeed, if the Atomos transactions are shrunk to match the scope of Java `synchronized` statements, a short-transaction version of Atomos can achieve a speedup of 25.5 on 32 CPUs. However, the long-transaction Atomos versions serve to show how a novice parallel programmer can iteratively refine a parallelization in a guided way to achieve a reasonable level of performance with obvious correctness at each step.

5.6 Conclusions

Semantic currency control allows concurrent access to data structures while preserving the isolation, and therefore serializability, properties of transactions. `TransactionalMap` and `TransactionalSortedMap` collection classes were built for this purpose using the concept of multi-level transactions built upon open nesting. `TransactionalQueue` showed how these ideas can be used to break the isolation property in structured ways when it is desired to trade serializability for performance. Such reusable collection classes should be part of the standard library of a transactional programming language such as Atomos.

While standard library classes are convenient for many programmers, a straightforward operational analysis and implementation guidelines allow programmers to safely design their own concurrent classes, in cases where they need to create new or augment existing data structures.

Chapter 6

Conclusion

> If I have not seen as far as others, it is because
> giants were standing on my shoulders.
>
> – Hal Abelson

In this dissertation I set out to prove the following thesis:

> If transactional memory is going to make parallel programming easier, programmers will need more than a way to convert critical sections to transactions.

In Chapter 3 I covered JavaT, a set of rules for the transactional execution of Java with a transactional memory system. I showed why simply treating critical sections as transactions could not maintain the expected behavior of programs involving monitors. The performance evaluation showed that a JavaT execution was competitive with fine-grained locks, and more importantly, that simpler coarse-grained transactions could perform as well as fined-grained locks in some cases.

In Chapter 4 I covered Atomos, a programming language designed for transactional memory. I presented the `watch` and `retry` constructs as a transactional alternative to monitors. I also showed the usefulness of going beyond simple transactions by showing the usefulness of open-nested transactions and transaction handlers for implementing applications, libraries, and the runtime system.

In Chapter 5 I covered semantic concurrency control and how it can be used to support a simpler programming model where most execution occurs in coarse-grained transactions. I showed how the Atomos primitives combined with multi-level transactions could be used to create transactional collection classes and demonstrated their use in a parallelization of a high-contention version of SPECjbb2000.

The following sections discuss directions for future work and then conclude with some final thoughts.

6.1 Future Work

Transactional memory is still a new and developing research area with many possible avenues for future work. In the following sections I will give future directions related to the work I have presented.

6.1.1 Programming Model

Chapter 4 briefly touched on supporting loop-level parallelism within the Atomos programming language. Syntactic support for loop-level parallelism along the lines of OpenTM transactional interface is well worth considering [9]. Loop-level parallelism can be an easy way to find parallel tasks for execution without requiring the program to explicitly use threads, although often its applicability is restricted to very regular code.

A promising area for future work is alternative programming models that allow programmers to more flexibly define potentially parallel computation. One such example is the recent work on Automatic Mutual Exclusion [65]. Another area related to the creation of parallel computation is allowing parallelism within a transaction as described in recent new semantic models [93].

6.1.2 I/O

Appendix C covers approaches to dealing with non-transactional operations called from within transactions. Many of the proposals discussed there have yet to be evaluated, and while there has been some work on characterizing workloads that contain I/O based on their sequential behavior, there has been little practical work in evaluating these approaches in transactional memory systems.

Some next steps in the development of a language like Atomos might include demonstrating the use of semantic concurrency control with B-trees, adding transactional aware versions of classes such as `BufferedOutputStream`, as well as a transaction aware version of a logging interface such as Java's `java.util.logging` or Apache log4j.

A larger project would be to make the Java Transaction Service (JTS) aware of transactional memory as a managed resource. This would allow coordinated access with other transactional resources, such as databases.

6.1.3 Garbage Collection

The interaction between transactional memory systems and garbage collectors has barely started to be explored. My experiments running on an architecture simulator do not run long enough to exercise the garbage collector, even though an individual experiment may run for more than a day of wall clock time. One line of future work could simply be to survey a selection of common garbage collector algorithms and categorize their behavior on a transactional memory system. Another possible area for future work is how transactional memory could be used to aid in the implementation of the garbage collector itself.

6.1.4 Database

Time and time again I have found solutions to problems facing transactional memory in the literature of the transactional database community. Some possible paths for exploration might include the use of savepoints in the implementation of transactional memory systems, as well as ways of exploiting parallelism within a single transaction.

6.2 Final Thoughts

I believe the true promise of transactional memory is in making parallel programming easier. This means transactional memory systems should be evaluated for their ability to run long transactions scalably, not by their ability to hold their own against code with very short transactions based on fine-grained locking parallelizations. Even if a system scales well with embarrassingly parallel applications or fine-grained transactions, it is also important to show scalability for applications with long-running transactions accessing shared data, since the ultimate goal is to make parallel programming easier by giving the programmer the performance of fine-grained locking while only using coarse-grained transactions.

Finally, I hope that my evaluation will convince the implementers of both hardware and software transactional memory systems of the benefits and need for rich transactional semantics. As the database community has shown, there is a lot more to transactional systems than simple atomicity.

Appendix A

Simulation Environment

per aspera ad astra

– Seneca the Younger

A.1 Simulated Architecture Details

All execution results in this dissertation come from an x86 CMP simulator that implements both a lazy versioning, optimistic conflict detecting hardware transactional memory system for evaluation's transactions and a MESI snoopy cache coherence system for evaluating locks. As noted in Section 3.3, this CMP HTM system does not support parallel commit. The HTM detects violations at the granularity of a cache line. More details on the simulated architecture can be found in [89, 90].

All instructions, except loads and stores, have a CPI of 1.0. The memory system models the timing of the L1 caches, the shared L2 cache, and buses. All contention and queuing for accesses to caches and buses is modeled. In particular, the simulator models the contention for the single data port in the L1 caches, which is used for processor accesses and commits for transactions or cache-to-cache transfers for MESI. A victim cache is used for recently evicted data from the L1 cache. Table A.1 presents the main parameters for the simulated CMP architecture.

More details on the instruction set architecture extensions for transactional memory, which were co-developed to support Atomos, can be found in [88].

Feature	**Description**
CPU	1–32 single-issue x86 cores
L1	64-KB, 32-byte cache line, 4-way associative, 1 cycle latency
L1 Victim Cache	8 entries fully associative
L2	512-KB, 32-byte cache line, 16-way associative, 3 cycle latency
L2 Victim Cache	16 entries fully associative
Bus Width	32 bytes
Bus Arbitration	3 pipelined cycles
Transfer Latency	3 pipelined cycles
Shared L3 Cache	8-MB, 32-byte cache line, 16-way associative, 20 cycle latency
Main Memory	100 cycle latency, up to 8 outstanding transfers

Table A.1: Parameters for the simulated CMP architecture. Bus width and latency parameters apply to both commit and refill buses. Shared L3 hit time includes arbitration and bus transfer time.

A.2 Signals

The simulator was modified to handle Unix signal delivery in a transactional system. Unix signals can be classified into two categories: synchronous and asynchronous. Synchronous signals are generated by the program itself when it performs an exception-causing operation, such as dividing by zero. Asynchronous signals come from outside the current program context, such as timer interrupts.

Because synchronous signals are caused by the current instruction, handlers run as part of the current transaction. This allows Java exceptions that rely on signals in the JikesRVM implementation such as `NullPointerException` to work as defined in Section 3.2.4. Asynchronous signals are less straightforward. If they are delivered immediately, logically the transaction buffers would have to be saved so that the unrelated signal handler could not interfere with the transaction that just happened to be interrupted. Non-`synchronized` transactions can just commit anywhere, allowing us to commit and execute the asynchronous signal handler in a new transaction. However, if all processors are executing `synchronized` transactions that must complete atomically, then delivery of the signal must be postponed until one of the transactions commits. This could result in unacceptably long interrupt-handling delays, however, so it may be necessary to just roll back a `synchronized` transaction to free up a processor immediately. In my implementation, the simple approach of delivering asynchronous signals on the first available non-transactional processor is followed.

A.3 Physical versus Virtual Addresses

The simulator does not differentiate between physical and virtual addresses. In a system with both physical and virtual addresses, applications programmers work with virtual addresses while cache coherence is typically maintained on physical addresses to avoid issues with different virtual addresses being aliases for the same physical address.

System software frequently has to deal with mappings between these two address domains. Mapping from virtual to physical address is easy as operating system software can consult page tables. This might need to be done to provide a phyiscal address as the target of a direct memory acesss (DMA) based on a user space pointer. However, mapping from physical to virtual addresses is not as straightforward, although it can be done in selected cases. For example, an unaligned memory access might be detected based on a physical address, but a signal handler needs to provide a virtual address to an application level handler. Some architectures use the current state of the processor to deduce the virtual address that causes a fault by inspecting the instruction pointer and register state.

Violation handlers discussed in Section 4.2.1 also can suffer from this physical to virtual address problem since conflict detection as part of the cache is based on physical addresses, yet a handler might need to work in terms of virtual addresses. However, a violation handler does not have the context of a faulting instruction to assist in the mapping from physical to virtual address since a load that caused a read dependency may have occured many cycles in the past. Generally, operatings systems do not have an efficient way to perform an arbitrary physical address to virtual address mapping.

Fortunately in the context of the Atomos scheduler implement, there is no need to solve the full physical to virtual address mapping problem. Violations are expected, either on well know queue addresses used to communicate with the scheduler thread or on watch addresses specified by the program. The runtime system can ask the operating system for the virtual to physical address mapping for queue addresses when they are created and for new watch addresses as needed, so that it can perform its own reverse mapping. The operating system would need to callback the runtime system if these mapping changes to keep the scheduler map coherent so that expected violations are not lost.

Appendix B

JikesRVM Implementation

The results are accurate.
– P. J. Denning [37]

In order to run Java-like programs with transactional semantics, it was first necessary to get a Java virtual machine to run within a transactional memory environment. As mentioned in Chapter 2, the Jikes Research Virtual Machine (JikesRVM) version 2.3.4 was chosen to evaluate Java programs. JikesRVM, formerly known as the Jalapeño Virtual Machine, has performance competitive with commercial virtual machines and is open source [7].

B.1 Running JikesRVM in a TM Environment

While getting applications to run transactionally with JavaT was straightforward, a number of system issues and virtual machine implementation details needed to be addressed to get a working system with good performance.

B.1.1 Scheduler

The JikesRVM scheduler multiplexes Java threads onto Pthreads. As Java threads run, code inserted by the compiler in methods prologues, epilogues, and back-edges determines if the current thread should yield control back to the scheduler. Unfortunately, when the scheduler notifies threads to yield, all running transactions are violated by the write. The scheduler also maintains a global wakeup queue of ready threads. Multiple threads trying to yield and reschedule can violate each other. While open-nested transactions could address the unwanted violations, they cannot help with the problem of context switching Java threads between two different Pthreads. It is not enough to simply move the register state, since the current transactional state also includes the buffer read- and write-set.

The simple workaround to the context switching problem was to avoid multiplexing threads by matching the number of Java threads to the number of Pthreads, maintaining per processor scheduler queues, and pinning threads to processors. The scheduler never needed to preempt the threads and could not migrate them to other processors, avoiding the above issues. A next step could be to defer the thread switch test until transaction commit time. Similar to signals, another solution could be to commit non-`synchronized` transactions at thread switch points and to rollback `synchronized` transactions.

B.1.2 Just-In-Time Compiler

JikesRVM features the Adaptive Optimization System (AOS), that is used to selectively recompile methods dynamically. When it first compiles a method, JikesRVM typically uses its baseline just-in-time (JIT) compiler that is geared to generating code fast rather than generating fast code. The generated code contains invocation counters to measure how often a method is called and edge counters to measure statistics about basic block usage. Based on these statistics, the most-used methods are selected for recompilation by an optimizing compiler that can use the basic block statistics to guide compilation.

Unfortunately, these global counters introduce data dependencies between transactions that are out of the application's control. A simple workaround is to just disable the adaptive optimizing system and force the use of the optimizing compiler at all times, although without statistics information. However, since these counters do not need to be 100% accurate, a simple solution might be to allow non-transactional updates of the counters. For example, if a load instruction could be non-transactional, it could be used to read counters in all generated code [61]. Since no transaction would consider the location to have been read as part of their transaction, violations would be avoided. The newly incremented value could then be written back normally as part of the surrounding transaction commit.

A more commonplace problem for all dynamic linkers, including JIT compilers and traditional C applications, is avoiding violations when resolving references. Two threads trying to dynamically link the same unresolved routine will both invoke the JIT and update the reference, causing one of the threads to violate. Again, using a non-transactional load to test the reference could avoid the violation, at the cost of the occasional double compilation or reference resolution. This becomes more important with systems like AOS, where high-usage methods are the ones most often recompiled at run time.

While some hardware transactional memory systems allow non-transactional reads and writes with transactions [61], *open-nested transactions* provide an alternative approach that preserves some atomicity transactional properties of transactions, while reducing isolation between transactions. [88, 24, 99, 96]. By placing runtime operations, such as method invocation counter updates, within open-nested transactions, the parent application transactions are not rolled back by conflicts involving only the child open-nested transaction.

B.1.3 Memory Management

Memory management is another area where Java virtual machines need to be aware of transactions. Poorly designed memory allocators can cause unnecessary violations. Early Java virtual machines had scalability problems because of contention for a single global allocator. While this has been addressed in current virtual machines intended for server use, shared structures still exist. For example, page allocation requests from a common free list cause data dependency violations. As a result, instead of refilling local allocators on demand, if a threads allocator is running low on pages, it may be better to refill it before entering a `synchronized` block.

Garbage collection can have issues similar to preemptive thread scheduling. Current stop-the-world collectors assume they can suspend threads, possibly using those threads' processor resources for parallel collection. As with thread scheduling, the garbage collector may want to commit or rollback all outstanding transactions. Concurrent collectors need to try to partition their root sets to avoid violations, and keep their own transactions small to avoid losing too much work when violations do occur. In the end, for this evaluation the garbage collector was disabled and applications were run with a two gigabyte heap, but transactional garbage collection is a definite area of future work.

B.1.4 Impact of Cache Line Granularity

When the JavaT results from Chapter 3 were first published [21, 20], the results were collected on a TCC simulator that could detect violations with word-level granularity [89]. However, when used to collect the results for this dissertation, as discussed in Appendix A, it detects violations at the granularity of a cache line. This means that a read and a write of different words by different transactions cause a violation, even though logically these reads and writes are to independent words.

When running with cache line granularity, careful Java array and object allocation layout can help minimize some of the worst problems from these *false violations*. For arrays, allocating the header, which includes the size and element type information, in a different cache line than the first elements can reduce violations. Having the size in a different cache line avoids violations between transactions reading the size and those writing elements at the start of the array that are in the same cache line as the size. Similarly, having the element type in a separate cache line reduces violations threads writing elements in the same cache line as the the element type and those reading the element type but writing to other cache lines. For objects, a similar split of object header and body into two cache lines can reduce some violations between writers to fields in the first cache line of the objects and other transactions that access the header for vtable and other similar information. However, it is worth noting that

Feature	**Transactional implementation**
Signals	synchronous signals delivered within transactions asynchronous signals delivered between transactions
Scheduler Context Switching	1.) rollback partial transactions on context switch 2.) delay context switch until end of transaction
Just-In-Time Compiler	1.) non-transactional update of shared runtime values 2.) open-nested update of shared runtime values
Memory Allocation	thread local pool refilled between transaction
Garbage Collector	rollback partial transactions before collection

Table B.1: Java virtual machine issues in a transactional memory system.

there is a trade-off between wasted space and reduced violations that probably only makes this worth while for larger objects and arrays that span several cache lines, since if the object is less than a cache line in size, there is no way to avoid violations between a read and write to different parts of the object.

It is also worth noting that this word-level versus cache-line granularity issue is a serious correctness problem for hardware implementations of "early release" in hardware transactional memory systems, as mentioned briefly in Chapter 4 [33, 119]. Typically, a programmer thinks of early release as a way to remove a word from the transactions read-set, but in an HTM system with cache line granularity, it is unclear what to do with such an instruction from the programmer. For correctness, the early release directive should be ignored since other words in the line may also have been speculatively read, but this would negatively impact the expected performance under contention when compared to a system with word granularity.

B.1.5 Summary of JikesRVM Issues and Changes

Table B.1 provides a summary of issues found in the Java virtual machine implementation.

All execution results in this dissertation were evaluated using JikesRVM with the following changes. The scheduler was changed to pin threads to processors and not migrate threads between processors. Methods were compiled before the start of the main program execution. The garbage collector was disabled and a two gigabyte heap was used. Results focus on benchmark execution time, skipping virtual machine startup.

B.2 Atomos

The following discusses additional details on building and running Atomos on top of JikesRVM.

B.2.1 Atomos Language Implementation

The Atomos programming language was implemented as a source-to-source transformation using Polyglot [102]. The transformation is fairly shallow, with the Atomos transactional constructs converted in a relatively straightforward way to use JikesRVM `VM_Magic` methods extensions. The `VM_Magic` methods are replaced by the baseline and optimizing compiler with simulator's HTM extensions to the x86 ISA.

B.2.2 Violations from Long Transactions

Running long-running transactions within Atomos found two new sources of violations within the underlying JikesRVM virtual machine. The first was from the `BumpPointer` allocator requests to the `MonotonePageResource`. The `MonotonePageResource` is a global allocator that provides pages to the per processor `BumpPointer` allocator. The second source of violations was from `ReferenceGlue.addCandidate` which keeps a global list of weak references, which was invoked because of the use of `ThreadLocal` storage. In both cases, accessing the global state from an open-nested transaction prevents this from being an unnecessary source of violations to applications.

B.2.3 Open Nesting and Stack Discipline

The use of `synchronized`-style block statements for nested transactions in Atomos is critical for correctness in TM systems that do not exclude the stack from the read- and write-sets. Without proper block nesting of transactions on the stack, a closed-nested transaction may roll back to find its stack modified by an open-nested transaction. For example, in a system with a library style interface for beginning and committing transactions, a program might create a new closed-nested transaction in a function and then return. If that transaction aborts, it will need the stack to be restored to its previous state so that the function can properly find values such as its return address on the stack. However, if after returning, the program commits an open-nested transaction that reuses an overlapping part of the stack, the closed-nested transaction may fail to find needed state on rollback, causing anything from incorrect program execution to an exploitable stack-smashing attack on the return address. This is very similar to the problem of *live stack overwrite* problem faced by TxLinux x86 interface handlers [108].

JikesRVM running on x86 provided an additional challenge to properly maintaining the stack in the face of open-nested transactions. When starting a transaction, transactional memory systems checkpoint register state to enable rollback of the program state. When initially running Atomos on a PowerPC-based simulator, this was sufficient for supporting open-nested transactions in the JikesRVM environment. However, on register starved x86, JikesRVM uses a calling convention that did not work with open-nested transactions without modification. The problem was that the frame pointer was not stored in a register, but instead stored in a thread local variable accessed indirectly through a register dedicated for processor specific state. If a thread only used closed-nested transactions, this frame pointer value, which is stored in memory, would be rolled back along with other memory state. However, if an open-nested transaction made a functional call and committed, it would write a new value for the frame pointer to memory that would not be rolled back if the parent closed-nested transaction was aborted. The solution is to have an abort handler for closed-nested transactions that restores the proper frame pointer on rollback.

Appendix C

Non-transactional Operations

Don't do that, then! :imp.
[from an old doctor's office joke about a patient with a trivial complaint] Stock response to a user complaint. 'When I type control-S, the whole system comes to a halt for thirty seconds." 'Don't do that, then!" (or 'So don't do that!"). Compare RTFM.

– The Jargon File 4.4.7

Transactions started out in the database world as a way to allow concurrent and possibly conflicting access to disk storage. Over time, the concept of transactions generalized to allow distributed transactions, potentially over a network, with a transaction manager coordinating multiple resource managers. This appendix will look at many different approaches to integrating transactional memory systems with non-transactional resources, such as disks and networks, that have traditionally been supported within transactional systems.

The approaches to integrating transactional memory and non-transactional operations can range from disallowing such operations, to allowing the programmer to potentially shoot themselves in the foot, all the way up to semantic concurrency control and distributed transaction systems. The following sections will work from the simplest proposals up, noting the pros and cons of each.

C.1 Forbid Non-transactional Operations in Transactions

The simplest approach to non-transactional operations in a transactional memory system is just to to forbid their use within transactions. This can be done either by reporting a runtime error [55] or at compile-time through use of a type system [56, 65]. The primary advantage of prohibiting non-transactional operations within transactions is simplicity. The major disadvantages are forcing programmers to partition their access from shared data from the I/O, as well as issues with composability of libraries that consist of both computation and I/O.

To take an example application, an HTTP server can read a request from the network interface into memory, use a transaction to process the request, producing a response to a buffer in memory, and finally write the response to the network after the transaction has committed. Moving the network I/O out of the transaction processing the request does not require a major restructuring of this code. However, since the actual processing of the request is within a transaction, it is restricted from doing other types of I/O such as reading files or talking to a database, leaving the server restricted to the serving responses based only on the input and the contents of memory.

C.2 Serialize Non-transactional Operations

Another relatively simple approach is to allow only one transaction at a time to perform a non-transactional operation. This thread becomes non-speculative and can no longer abort, even at its own request, since there is no way to undo the non-transactional operation it has performed. In addition, any contention between the non-speculative thread and other transactions must be decided in favor of the non-speculative thread. The advantage of this approach is the simplicity. The disadvantages are the serialization caused by transactions performing non-transactional operations and the lack of user directed abort.

To continue the HTTP server example, no I/O would be allowed to happen while processing a request. A typical request for a static file could read the file and return its contents to the request, potentially reading the contents into an in-memory cache to avoid serializing on requests to that file in the near future. However, if the server had little locality in its file requests, the requests would be serialized as each request took turns accessing the disk within their transactions.

C.3 Allow Selected Non-transactional Operations

Another approach is for non-transactional operations to serialize transactions in the common case, but then allow selected classes of operations and specially marked instances of operations as not causing serialization. The advantage of this is removing some unnecessary serialization when it is unneeded with the disadvantage of allowing programmers to incorrectly perform non-transactional operations with no way to undo their effects.

In the HTTP server example, consider the use cases of `gettimeofday` and debug logging. The `gettimeofday` system call might be used to check if cached data is still valid or to expire old data. Since it does not modify any non-memory state, it is reasonable to run it without serializing on the calling transaction. For debug logging, any tracing of the request processing is desired regardless of the transaction outcome. Since this information is simply for the programmers use in diagnosing issues, it is okay that there is no way to undo it if the calling transaction aborts in the future. Note that not all output should happen non-transactionally. While the instances of debug output should be allowed, other file output should still cause serialization.

C.4 Use Transaction Handlers to Delay or Undo Operations

Another approach to dealing with non-transactional operations is through the use of transaction handlers, as discussed previously in Section 4.1.4. Commit handlers offer a way to delay non-transactional operations until after a transaction completes. Abort handlers allow a compensating action to be performed if a transaction fails, allowing transactions to perform I/O in a transaction, but giving a structured way to undo its effects if necessary. The great advantage of transaction handlers is the flexibility they give to the programmer, but at the same time the correct design compensating actions also adds to the burden of programmers.

There is some hope that some common patterns of using commit handlers could be implemented by library authors rather then the application programmer directly. Non-transactional operations within transactions are recorded at each call but only performed at commit time using transaction callbacks, a common solution in traditional database systems [63]. For example, a reusable wrapper class could be created for the common case of buffered stream output, to automatically delay output until commit time. In a transactional runtime, this functionality could be built into the standard Java classes like `BufferedOutputStream` and `BufferedWriter` classes, allowing common code to work without modification.

To return to the HTTP server example, transaction handlers could be used to remove more cases of serialization from file and network I/O. Consider the processing of a file upload via the HTTP POST method. The file could be uploaded to a temporary location as part of the transaction. If the transaction commits, the temporary file can be moved to its permanently location. If the transaction aborts, the temporary file can be deleted. In perhaps a more important case, the request could now be allowed to talk over the network to a database to perform a single database transaction. The final commit or abort of the database transaction can be tied to the success or failure of the memory transaction.

C.5 Semantic Concurrency Control

The approach of semantic concurrency control can be applied to abstract data types involving non-transactional operations. A logical extension of the previous approach, a `TransactionalMap` could be used around a `Map` implementation that provides durable storage, such as a B-tree. The advantage of this approach is that programs can use existing semantic concurrency control implementations, while the disadvantage is that not all non-transactional operations have well defined inverses to allow perfect compensating actions. To use a common database example, if the non-transactional operation is to launch a missile, the compensating action might be to have the missile self-destruct, but it does not return us to the original state of the system.

To continue the HTTP server example, the incoming requests could use a set of B-trees protected by semantic concurrency control to implement a small database. Note that a transaction involving operations involving multiple trees would have full ACID transaction semantics, even though the B-tree implementations all operate independently of each other.

C.6 Distributed Transactions

The final approach to discuss is making the transactional memory system simply one of many resource managers. The memory transactions could then interact with other transactional resource manager, such as file systems, databases, and message queues. Given commodity OS for transactional file systems [103], handling the common case file operations at a system wide level rather than with *ad hoc* approaches within the transactional memory implementation seems like a good solution. However, the downside is the potential performance downside of interacting with the system transaction coordinator, although this can be avoided in the potentially common case of memory only transactions.

In the HTTP server example, request processing could interact with a transaction file systems and databases without any special handlers or concurrency control outside the transactional memory implementation.

C.7 Summary of Approaches

Modular systems are built on composable libraries. What is needed are general ways to address non-transactional operations without disturbing the logical structure of the code. However, different types of non-transactional operations may need different approaches.

To finish the HTTP server example, many of these approaches may end up working together. Distributed transactions could address the common problems with access to files and transactional resources over the network. Semantic concurrency control might be used to implement durable resource managers within the transactional memory system. Commit handlers might provide ways of coping with output to peripherals that lack undo such as printers, ATM machines, and missiles. Certain operations such as `gettimeofday` can simply be allowed to run without any restrictions from the system. Finally, any remaining non-transactional operations, such as process creation, can be handled by serializing the call transaction.

C.8 Related Work

C.8.1 No I/O in Transactions

Harris et al. integrated TM into Concurrent Haskell to take advantage of Haskell's type system, which identifies I/O operations as part of signatures [56]. This allows static checking to prevent non-transactional operations with atomic regions.

Isard and Birrell introduced Automatic Mutual Exclusion, which is a programming model where non-transactional operations can only occur in unprotected regions that are outside of the transactional execution environment [65].

C.8.2 Serialization

Hammond et al. allowed a thread that needed to perform a non-transactional operation to take and hold a commit token to ensure that only one transaction could perform a non-transactional operation at a time [53, 50]. Blundell et al. evaluated a similar system that allows one unrestricted transaction to perform non-transactional operations [13]. Both of these approaches are similar to the non-speculative thread in TLS and related systems discussed in Section 2.5.2 and Section 2.5.3.

C.8.3 Transaction Handlers

Harris explored how the `ContextListener` listener, a form of transaction handler discussed in Section 4.1.4, [54] could be used to defer output until after transactions commit as well as rewind input on transaction abort.

McDonald et al. presented the HTM semantics for a system with support for transaction handlers, demonstrating how I/O could be deferred until transaction commit [88]. This was the system used for the original Atomos evaluation [24].

Baugh and Zilles did a study of the use of system calls in a desktop application, the Firefox web browser, as well as a server application, the MySQL database, to estimate the effectiveness of some of the proposed approaches to non-transactional operations [10]. They found that some system calls required no compensation and that many could be compensated for by another syscall, while others might require some sort of higher level knowledge of the program behavior to provide compensation. Their analysis showed that simple deferral of system calls until a commit handler would not work in many cases since callers depended on the results of the system call. They also found that approaches that serialize on transactions could seriously hurt the performance of the studied application.

C.8.4 Distributed Transactions

Carlstrom et al. discussed integrating transactional memory systems with distributed transaction environments [22]. The previously mentioned Baugh and Zilles work also discussed how integrating with a transactional file system could address compensation for many file system system calls [10].

The Apache Commons Transaction Component provides resource managers for `Map` classes as well as for file access [43]. Similar to the transactional collection class approach of Chapter 5, programmers use a wrapper class that implements standard interfaces [23]. Three different wrappers are provided: a basic `TransactionalMapWrapper`, an `OptimisiticMapWrapper`, and a `PessimisticMapWrapper`. Transactional file access is allowed via standard `InputStream` and `OutputStream` interfaces. Note that Commons Transaction is not related to transactional memory, but allows an application to tie memory and file operations to a transaction outcome that include other transactional resources such as a database.

C.8.5 Memory Mapped I/O

Kuszmaul and Sukha explored performing file I/O in a page-based STM system by using memory mapped files [79]. This meant that file I/O did not need to be treated differently than memory operations and also meant that a conflicting file I/O between multiple threads were caught by the TM system. While this technique is applicable to file I/O, it is not useful in general non-transactional operations.

C.8.6 OS Work

Ramadan et al. approached transactional memory and system calls from a different perspective, that of using transactional memory within the Linux kernel syscall implementation [108]. Their MetaTM is an HTM system with special features to support implementing x86 interrupt handlers for their TxLinux kernel. Rossbach et al. explains how the TxLinux kernel uses HTM, particularly in its scheduler implementation [110].

Bibliography

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 63–74, New York, NY, USA, 2008. ACM.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

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

[9] 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.

[10] L. Baugh and C. Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *2008 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2008.

[11] 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.

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

[13] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, April 2006.

[14] 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.

[15] H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268, New York, NY, USA, 2005. ACM Press.

[16] E. Brevnov, Y. Dolgov, B. Kuznetsov, D. Yershov, V. Shakin, D.-Y. Chen, V. Menon, and S. Srinivas. Practical experiences with Java software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 287–288, New York, NY, USA, 2008. ACM.

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

[18] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial*. Addison-Wesley Professional, third edition, January 2000.

[19] 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.

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

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

[22] B. D. Carlstrom, J. Chung, C. Kozyrakis, and K. Olukotun. The software stack for transactional memory: Challenges and opportunities. In *First Workshop on Software Tools for Multi-Core Systems*. March 2006.

[23] 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.

[24] 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.

[25] 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.

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

[27] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

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

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

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

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

[32] C. Click. A lock-free hashtable. JavaOne Conference, May 2006.

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

[34] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in C++. In *TRANSACT '07: Second ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, August 2007.

[35] 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.

[36] J. Danaher, I.-T. Lee, and C. Leiserson. The JCilk Language for Multithreaded Computing. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. University of Rochester, October 2005.

[37] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

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

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

[40] R. Ennals. Efficient software transactional memory. Technical Report IRC-TR-05-051,, Intel Research Cambridge, 2005.

[41] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: a distributed transaction facility.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991.

[42] C. Flanagan. Atomicity in multithreaded software. In *Workshop on Transactional Systems*, April 2005.

[43] T. A. S. Foundation. *package org.apache.commons.transaction.* `http://commons.apache.org/transaction`, January 2005.

[44] Free Software Foundation, *GNU Classpath 0.18*. `http://www.gnu.org/software/classpath/`, 2005.

[45] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, New York, NY, USA, 1987. ACM Press.

[46] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154. IEEE Computer Society, 1981.

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

[48] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *MSPC: Workshop on Memory Systems Performance and Correctness*, October 2006.

[49] 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.

[50] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *Micro's Top Picks, IEEE Micro*, 24(6), Nov/Dec 2004.

[51] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13, New York, NY, USA, October 2004. ACM Press.

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

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] M. Herlihy. SXM: C# software transactional memory. `http://research.microsoft.com`, May 2006.

[59] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006. ACM Press.

[60] 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.

[61] 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.

[62] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC: Workshop on Memory Systems Performance and Correctness*, October 2006.

[63] IBM Corporation. *Encina Transactional-C Programmer's Guide and Reference for AIX, SC23-2465-02*, 1994.

[64] International Organization for Standardization. *ISO/IEC 9075-5:1999: Information technology —- Database languages — SQL — Part 5: Host Language Bindings (SQL/Bindings).* International Organization for Standardization, Geneva, Switzerland, 1999.

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

[66] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, August 2005.

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

[68] *Java Specification Request (JSR) 133: Java Memory Model and Thread Specification*, September 2004.

[69] *Java Specification Request (JSR) 166: Concurrency Utilities*, September 2004.

[70] M. Jordan and M. Atkinson. *Orthogonal Persistence for the Java Platform.* Technical report, Sun Microsystems, October 1999.

[71] 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.

[72] S. Kapil. UltraSparc Gemini: Dual CPU processor. In *Conference Record of Hot Chips 15 Symposium*, Palo Alto, CA, August 2003.

[73] A. Kimball and D. Grossman. Software transactions meet first-class continuations. In *Scheme and Functional Programming Workshop '07*, September 2007.

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

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

[76] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, 48(9):866–880, September 1999.

[77] M. Kulkarni, L. P. Chew, and K. Pingali. Using Transactions in Delaunay Mesh Generation. In *Workshop on Transactional Memory Workloads*, June 2006.

[78] 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.

[79] B. C. Kuszmaul and J. Sukha. Concurrent cache-oblivious B-trees using transactional memory. In *Workshop on Transactional Memory Workloads*, Ottawa, Canada, June 2006.

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

[81] D. Lea. *package util.concurrent*. `http://gee.cs.oswego.edu/dl`, May 2004.

[82] S. Liang. *Java Native Interface: Programmer's Guide and Reference.* Addison-Wesley Longman Publishing Company, Inc., 1999.

[83] B. Liblit. An operational semantics for LogTM. Technical Report CS-TR-2006-1571, University of Wisconsin-Madison, Department of Computer Sciences, August 2006.

[84] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, 1983.

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

[86] 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.

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

[88] 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.

[89] 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.

[90] A. N. McDonald. *Architectures for Transactional Memory.* PhD thesis, Stanford University, 2008.

[91] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, New York, NY, USA, 1996. ACM Press.

[92] 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.

[93] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 51–62, New York, NY, USA, 2008. ACM.

[94] 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.

[95] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, April 1981.

[96] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, March 1985.

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

[98] J. E. B. Moss, N. D. Griffeth, and M. H. Graham. Abstraction in recovery management. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 72–83, New York, NY, USA, 1986. ACM Press.

[99] 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.

[100] J. D. Newton. *Uncommon Friends: Life with Thomas Edison, Henry Ford, Harvey Firestone, Alexis Carrel & Charles Lindbergh.* Harcourt, 1987.

[101] 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.

[102] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In G. Hedin, editor, *CC 2005: International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, April 2003.

[103] J. Olson. NTFS: Enhance your apps with file system transactions. *MSDN Magazine*, July 2007.

[104] W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

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

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

[107] 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.

[108] 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.

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

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

[111] 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.

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

[113] B. Sandén. Coping with Java threads. *IEEE Computer*, 37(4):20–27, 2004.

[114] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

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

[116] M. Sherman. Architecture of the encina distributed transaction processing family. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 460–463, New York, NY, USA, 1993. ACM Press.

[117] O. Shivers and B. D. Carlstrom. *Scsh Reference Manual*, 0.3 edition, December 1994. Current version of manual available at `http://scsh.net`.

[118] 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.

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

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

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

[122] STAMP: Stanford transactional applications for multi-processing. `http://stamp.stanford.edu`.

[123] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[124] I. L. Trager. Trends in systems aspects of database management. In *Proceedings of the 2nd International Conference on Databases*. Wiley & Sons, 1983.

[125] M. Tremblay. Transactional memory for a modern microprocessor. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 1–1, New York, NY, USA, 2007. ACM.

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

[127] J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A semantic framework for designer transactions. In D. A. Schmidt, editor, *Proceedings of the European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 249–263. Springer-Verlag, April 2004.

[128] 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.

[129] W. Weihl and B. Liskov. Specification and implementation of resilient, atomic data types. In *SIGPLAN '83: Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues in software systems*, pages 53–64, New York, NY, USA, 1983. ACM Press.

[130] G. Weikum and H.-J. Schek. Architectural issues of transaction management in multi-layered systems. In *VLDB '84: Proceedings of the 10th International Conference on Very Large Data Bases*, pages 454–465, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

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

[132] P. T. Wojciechowski. Isolation-only transactions by typing and versioning. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on*

Principles and practice of declarative programming, pages 70–81, New York, NY, USA, 2005. ACM.

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