12-2016

Hybrid STM/HTM for nested transactions in Java

Keith G. Chapman
Purdue University

HYBRID STM/HTM FOR NESTED TRANSACTIONS IN JAVA

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Keith G. Chapman

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2016

Purdue University

West Lafayette, Indiana

**PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance**

This is to certify that the thesis/dissertation prepared

By Keith Godwin Chapman

Entitled

Hybrid HTM/STM for Nested Transactions in Java

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Altony L Hosking Chair	Walid G. Aref
J. Eliot B. Moss	
Mathias Payer	
Tiark Romph	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Antony L. Hosking

Approved by: Sunil Prabhakar / William J. Gorman 12/5/2016

Head of the Departmental Graduate Program Date

To my parents, wife, daughter and son.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Professor Antony Hosking for his invaluable guidance and advice through my PhD studies. I would also like to thank Professor Eliot Moss for his input and advice through the project. After Professor Hosking moved to Australia we moved out weekly hangout to 9 PM eastern time to which Professor Moss attended without fail. I appreciate your commitment to making this project a success.

Also I would like to thank the members of my advisory committee, Professor Mathias Payer, Professor Tiark Rompf and Professor Walid Aref for agreeing to serve.

Many people have helped me through this endeavor. My lab mate and good friend Ahmed Hussein has helped me with word and deed throughout my stay at Purdue, for which I'm ever so grateful. A special note of thanks goes out to Dr. Sanjiva Weerawarana for motivating me to pursue graduate studies. I'd like to thank Dr. David Grove and Dr. Vijay Saraswat at IBM T.J Watson research center, Dr. Jan Rellermeyer and Dr. John Carter from IBM Research, Austin, TX and Dr. Herman Venter from Microsoft Research, Redmond, WA for supervising and mentoring me during my various internships and giving me advice.

While at Purdue, I maintained my sanity largely with the help and support of the wonderful Sri Lankan community at West Lafayette. I'd also like to extend my gratitude to all the members of the First United Methodist Church and the Wesley Foundation for been my family away from home.

Last but not least I would like to thank my wife Madara, daughter Hazelle, son Tristan, my parents and my sister for your love and patience. Non of this would have been possible without the many sacrifices you've made.

PREFACE

In all chapters and related publications of the thesis, my contributions are: researching background knowledge and related work; design and implementation of the XJ language; implementation of the XJ framework; conducting experiments; and writing and polishing the writing. My co-authors supported me in refining the ideas and design, pointing me to missing related work, providing feedback on earlier drafts, and polishing the writing. The XJ language extends the work of Ni et al. [42] while the XJ rewriting framework builds on the work of McGachey et al. [39]

All of the work presented henceforth has been published at peer reviewed conferences as follows.

1. K. Chapman, A. L. Hosking, J. E. B. Moss, and T. Richards. Closed and open nested atomic actions for Java: Language design and prototype implementation. In *International Conference on the Principles and Practice of Programming on the Java platform: virtual machines, languages, and tools*, PPPJ, pages 169–180, Cracow, Poland, Sept. 2014. doi: 10.1145/2647508.2647525 (Chapter 3)

2. K. Chapman, A. L. Hosking, and J. E. B. Moss. Hybrid STM/HTM for nested transactions on OpenJDK. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 569–585, Amsterdam, The Netherlands, Oct. 2016. doi: 10.1145/2983990.2984029. Distinguished Paper Award (Chapters 4 and 5)

3. K. Chapman, A. L. Hosking, and J. E. B. Moss. Extending OpenJDK to support hybrid STM/HTM: Preliminary design. In *ACM SIGPLAN Workshop on Virtual Machines and Intermediate Languages*, VMIL, Amsterdam, The Netherlands, Oct. 2016. doi: 10.1145/2998415.2998417 (Chapter 6)

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

API	Application Programming Interface
ASF	Advanced Synchronization Facility
AST	Abstract Syntax Tree
CPU	Central Processing Unit
DSTM	Dynamic Software Transactional Memory
HLE	Hardware Lock Elision
HTM	Hardware Transactional Memory
HyTM	Hybrid Transactional Memory
I/O	Input/Output
JIT	Just In Time
JNI	Java Native Interface
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tool Interface
LTM	Large Transactional Memory
PhTM	Phased Transactional Memory
RTM	Restricted Transactional Memory
SDE	Software Development Emulator
STM	Software Transactional Memory
TCC	Transactional Coherence and Consistency
TLR	Transactional Lock Removal
TM	Transactional Memory
TSX	Transactional Synchronization Extensions
UTM	Unbounded Trans- actional Memory
VM	Virtual Machine

VTM	Virtual Transactional Memory
XJ	Transactional Java

ABSTRACT

Chapman, Keith G. Ph.D., Purdue University, December 2016. Hybrid STM/HTM for Nested Transactions in Java. Major Professor: Antony L. Hosking.

Transactional memory (TM) has long been advocated as a promising pathway to more automated concurrency control for scaling concurrent programs running on parallel hardware. Software TM (STM) has the benefit of being able to run general transactional programs, but at the significant cost of overheads imposed to log memory accesses, mediate access conflicts, and maintain other transaction metadata. Recently, hardware manufacturers have begun to offer commodity hardware TM (HTM) support in their processors wherein the transaction metadata is maintained "for free" in hardware. However, HTM approaches are only *best-effort*: they cannot successfully run all transactional programs, whether because of hardware capacity issues (causing large transactions to fail), or compatibility restrictions on the processor instructions permitted within hardware transactions (causing transactions that execute those instructions to fail). In such cases, programs must include failure-handling code to attempt the computation by some other software means, since retrying the transaction would be futile.

This dissertation describes the design and prototype implementation of a dialect of Java, XJ, that supports closed, open nested and boosted transactions. The design of XJ, allows natural expression of layered abstractions for concurrent data structures, while promoting improved concurrency for operations on those abstractions. We also describe how software and hardware schemes can combine seamlessly into a hybrid system in support of transactional programs, allowing use of low-cost HTM when it works, but reverting to STM when it doesn't. We describe heuristics used to make this choice dynamically and automatically, but allowing the transition back to HTM opportunistically. Both schemes are compatible to allow different threads to run concurrently with either mechanism, while preserving transaction safety. Using a standard synthetic benchmark we demonstrate that HTM offers significant acceleration of both closed and open nested transactions, while yielding parallel scaling up to the limits of the hardware, whereupon scaling in software continues but with the penalty to throughput imposed by software mechanisms.

1 INTRODUCTION

The XJ language provides full blown support for closed, open nested and boosted transactions. The hybrid transactional memory system supported by the XJ framework allows hardware transactions and software transactions to proceed concurrently. Open nesting increases the envelope of concurrency and transaction sizes that can be accommodated in hardware.

Transactional memory (TM) allows programmers to group memory operations into *transactions* that appear to execute atomically: no transaction sees the intermediate states of other transactions executing in other threads, and all work of a transaction either happens (the transaction *commits*) or not (the transaction *aborts*). Transactional memory is more abstract than locking, and avoids many of the problems encountered with locks, such as deadlock, priority inversion, convoying, pre-emption, and reduced concurrency.

Transactional memory systems track memory read and write operations performed against disjoint memory units. When two transactions access the same memory unit and at least one of the accesses is a write then there is a *conflict*: one of the transactions must abort (discarding its pending writes) and restart. The transaction system must also manage atomicity: either all of a transaction's writes occur, or none of them, and to other transactions the writes appear to occur all at a single instant in time.

Software transactional memory (STM) systems track memory accesses in software, usually at the logical level of fields or objects of a host programming language. The overhead of this software instrumentation results in loss of throughput for memory accesses. Nevertheless, STM systems can still scale better than non-transactional synchronization schemes (such as locking) because of increased concurrency.

In contrast, hardware transactional memory (HTM) systems track memory accesses in hardware at the physical level of bytes, words, or cache lines, with little or no throughput overhead. However, current HTM proposals and implementations such as Intel's Transactional Synchronization Extensions (TSX), IBM's System Z, and AMD's Advanced Synchronization Facility (ASF), offer only *best-effort* hardware transactions: they can fail even when there are no conflicts [14; 16; 19; 30; 51; 56]. Such reasons for failure include lack of hardware capacity to track accesses, compatibility restrictions on instructions permitted to execute with transactions, page faults, and other hardware interrupts. As a result, HTM systems require software to take over when a hardware transaction cannot profitably be retried. For example, *hardware lock elision* (HLE) replaces lock regions with transactions, retrying some number of times in the case of conflicts, but falling back to lock acquisition when HTM otherwise fails [17; 48].

Proponents of transactional memory have long devised various models for aggregating nested execution of atomic actions into larger transactions. Most systems (including existing commercial HTM) simply fold the operations of nested transactions into the top-level outermost transaction, forming one large *flat* transaction. In this case, any conflict arising in a nested transaction will cause the top-level transaction to abort, discarding all of its effects. However, some systems allow nested transactions to abort independently of the parent while preserving the parent's atomicity, avoiding the loss of work performed by the parent due to a conflict by the child. *Closed* nesting [40] still aggregates the effects of nested transactions into their parent on commit, but allows retry of a nested transaction when it aborts, without necessarily aborting the parent. Other work [41; 42] has proposed *open* nesting as an extension to closed nesting, allowing improved concurrency at the cost of some programmer effort. This approach relies on having programmers annotate open transactions with an abstract *undo* action that can be used by the parent to revert the effects of the child if the parent aborts. The nested atomic actions still execute as transactions, with the usual conflict detection for their memory operations to ensure atomicity, but when they commit their memory effects become permanent and globally visible. The undo action allows their effects to be rolled back if the parent aborts. Open nesting also requires *abstract* *concurrency control*, so as to detect abstract conflicts between transactions that occur at the level of the abstract operations encapsulated by the open nested transactions.

1.1 Overview

We describe the design and prototype implementation of a dialect of Java, XJ, that supports a range of transactional programming abstractions, including open/closed nested transactions, and transactional boosting. This dissertation shows the extent to which these alternatives for nesting transactions can be accelerated by using HTM where possible, to avoid the respective overheads of their STM implementations, while allowing fallback to STM execution when the hardware fails to provide. In particular, we desire a system that presents a full-blown general transactional programming framework for Java, while automatically and dynamically choosing when to use HTM versus STM, and where hardware and software variants can execute concurrently and seamlessly while preserving transaction semantics.

The dissertation is structured as follows:

- The remainder of this Introduction chapter gives an overview of the XJ framework, and enumerates the contributions of this work.
- Chapter 2 discusses the prior work in this area, and summarizes some closely related fields.
- Chapter 3 details the language extensions we propose and discusses the syntax and semantics of the XJ language.
- Chapter 4 outlines the XJ framework that provides the prototype implementation. We discuss the major components of the XJ architecture and its implementation.
- Chapter 5 provides an evaluation of the XJ framework showing how open nesting increases the envelope of concurrency for HTM.

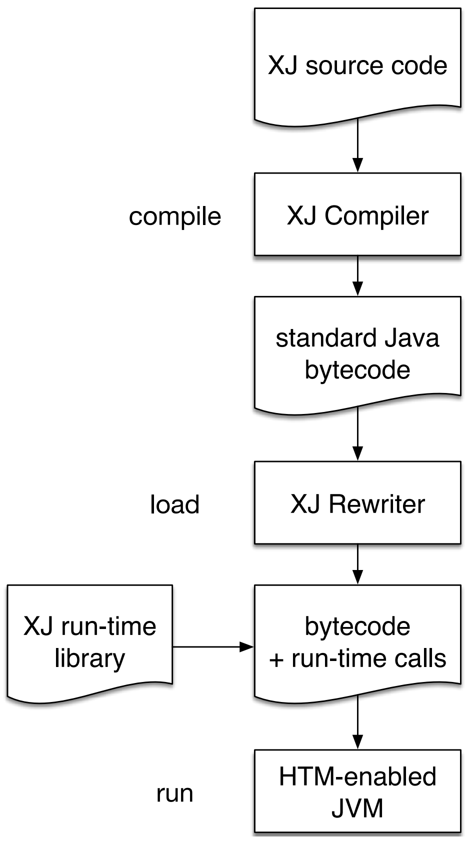


Figure 1.1.: XJ Tool Chain

- Chapter 6 discuss future work, detailing how the XJ framework can be integrated further into a production VM.
- Finally, Chapter 7 summarizes the dissertation.

1.2 XJ Framework

The XJ Framework consists of many integrated components as shown in Figure 1.1. It consists of a compiler that compiles XJ source code to standard java bytecode, a bytecode rewriting framework that instruments classes at load time, a runtime library that provides the functionality to track transactions and abstract locks, and a minimally modified JVM to provide HTM support. If run on an unmodified JVM, XJ could still function with its STM capabilities.

1.3 Principles and Approach

Transactions are usually described in terms of read and write operations performed against disjoint memory locations. Hardware and software transactional memory work in terms of either hardware memory units such as bytes, words, or cache lines, or, when incorporated into a programming language, in terms of variables/fields or objects. Regardless of the memory units in play, the transaction mechanism tracks reads and writes of those units to detect conflicts (two transactions access the same unit and at least one of them writes it) and manage atomicity (either all of a transaction's writes occur, or none of them, and they appear to occur at a single instant in time).

Here we will describe a design for Java that performs conflict detection on the unit of objects, and that tracks writes at the level of object (and static) fields. We adopt pessimistic concurrency control for objects modified by a transaction (i.e., writing requires acquiring a lock) and handle atomicity of update by allowing updates in place and undoing a transaction's uncommitted writes if the transaction aborts. Thus we will use an undo log. In principle any of these decisions could be varied; some would have a degree of visible impact on the language design, though much would remain the same. While we agree that no particular transaction management policy offers the best performance under all workloads, this seems to be a reasonable "middle of the road" choice. Nevertheless, we have designed our prototype implementation to allow future experimentation with alternative approaches.

What we just described briefly characterizes flat, *non-nested*, transactions. Here is a correspondingly short description of closed nesting. A closed nested transaction is either *top-level*, or a *child* subtransaction nested within a *parent* transaction. Logically, transactions accumulate read and write sets, which determine *conflicts* as well as what writes become visible upon commit.

Updates become globally visible only when a top-level transaction commits. When a transaction reads a value, it sees the value in its own read or write set (if there is one), otherwise the value seen by its parent. A top-level transaction will see the latest (globally committed) value, subject to subsequent overriding of the parent's values when the child commits, as follows.

When a nested transaction commits, its read and write sets merge with its parent's, the child's writes overriding any previous value in the parent. When a top-level transaction commits, its writes become permanent. When a transaction aborts, its read and write sets (and associated updates) are discarded or *rolled back*.[1] A transaction can succeed only if it has no conflicts with other transactions. Nested transactions refine the earlier definition of conflict (two transactions both accessing the same unit, at least one of them writing it) to say that there is conflict only when neither transaction is an ancestor of the other. In case of conflict, either or both must abort to prevent violation of transaction semantics, in which a legal execution is equivalent to a serial execution of the committed transactions (only), in some order (i.e., *serializability* [44]).

Nested transactions allow decomposition of a large transaction into smaller subtransactions, each of which can attempt some portion of work, and possibly fail (and perhaps be retried) without aborting work already accomplished by the parent. However, the parent still accumulates the read and write sets of all of its committed children (so writes by a child become visible to other unrelated transactions only when the top-most ancestor commits). Thus, as large transactions accumulate ever larger read and write sets from their children they will become more prone to failure due to higher probability of conflict. These failures reduce system throughput (the effective degree of concurrency).

Open nesting allows *further* increases in concurrency [42], by releasing concrete resources (e.g., memory reads and writes) earlier and applying conflict detection (and roll back) at a higher abstraction level. For example, transactions that increment and decrement a shared memory location would normally conflict, since they write to the same location. But, since increment and decrement commute as abstract operations, they can be implemented correctly with open nesting. An increment (say) does: read, add-one, write. The open nested transaction would be over and the updated field would not be part of the parent

[1] If the system performs updates in place and keeps an undo log, on commit of a child transaction the child's undo log is appended to the parent's. Thus, abort of the parent will remove the effects of the child *and* any other preceding effects recorded earlier in the parent's undo log.

transaction's read or write set. However, if the parent later aborts, it needs to run a compensating decrement to roll back the logical effect of the committed open nested transaction.

The only difference between open and closed nesting in terms of the read/write set execution model concerns what happens when a transaction commits. When an open nested transaction commits, it discards its read set, and commits its writes globally *at top level*.[2]

To support moving conflict detection from the concrete to the abstract level, when the committing open nested transaction releases its concrete memory resources (i.e., its memory reads and writes), it must typically claim some (set of) *abstract* resource(s) ("abstract locks") and provide a corresponding abstract compensation operation (e.g., the decrement in the earlier example) for use by its ancestors if they need to abort and roll back.

Prior work [42] showed that in some cases open nesting can greatly increase concurrency. However it does place more of burden on programmers who use it, since they (a) need to get the compensating actions right, and (b) likewise need to provide suitable abstract concurrency control. It has also been observed that if open and closed nesting are ever applied to the *same object*, deadlocks can occur that block both the completion of an open nesting action and a compensating action needed to abort the ongoing transaction.

If we view transaction conflicts and rollback in terms of *operations*, we can see greater similarity between closed and open nesting and highlight better the essential difference. Closed nesting works in terms of *read* and *write* operations, with the usual conflict rules on those operations. The undo of a *write* is a *write* that installs the original value of the memory unit. In the open nesting case we have a programmer-defined set of operations, with programmer-defined conflict rules and programmer-supplied rollback operations for each forward operation. So the essential difference when viewed from "outside" the transaction is the set of operations over which the transaction operates.

[2]It further discards its written data elements from the read and write sets of all other transactions. Given the conflict rules, these can only be its ancestors (it cannot commit unless those other unrelated conflicting transactions also abort). Well-structured programs respecting proper abstraction boundaries (not manipulating the same state at different transaction levels) will avoid this situation, but the rule makes the commit global, as intended.

However, the more abstract[3] transactions provided by open nesting—which offer increased concurrency because abstract concurrency control captures the essential semantic conflict while read/write level conflict detection over-estimates conflicts—must be built from *something*, and the individual operations must still appear to execute atomically. More precisely, they must be *linearizable* [27]: they must appear to occur at a single instant of time. Transactions are one way to achieve that linearizability, so it is natural to *implement* an open nested transaction using much the same mechanism as for closed nesting.[4]

1.4 Contributions

While the concept of transactional memory has been around for decades our approach differs in several ways from previous work (as described in chapter 2). This dissertation presents several main contributions:

XJ Language. We refine earlier proposals for open nesting constructs to combine open nested classes with a rich range of abstract locks, used to represent the abstract resources acquired by open nested transactions as they commit. Earlier approaches to adding open nesting to a programming language were vulnerable to a kind of internal deadlock that could prevent both forward progress of a transaction and successful undo to abort and remove the transaction. The design presented here solves that problem, overcoming a key reliability concern.

XJ Framework. We present a full-fledged implementation of extended Java language abstractions for *nested* transactions (where children can fail independently of parents, as opposed to the flattening of other systems) to gain improved concurrency; we call this system XJ (for transactional Java).

[3]We mean "abstract" in that conflicts don't occur at the physical level.

[4]Transactional boosting [24], however, recognizes that *how* that linearizability is achieved does not matter, and thus naturally supports an approach where existing non-transactional code is extended with transactional wrappers. It still needs abstract concurrency control of some kind, etc.

Hybrid TM design. We describe a hybrid HTM and STM scheme for nested (and boosted) transactions in Java, allowing HTM and STM to execute concurrently and compatibly.

Simple but effective back-off scheme. We demonstrate that are simple but effective back-off scheme make use of HTM where possible with adaptive seamless reversion to STM where not, for good performance.

Implemented on a high-performance JVM. We show how the hybrid TM design can be made to execute under optimized compilation with the high-performance (HotSpot-based) OpenJDK for Java, with minimal modification to the HotSpot compilers to add HTM intrinsics.

Performance study. We demonstrate via experiments using an established benchmark that HTM can significantly boost throughput and that falling back to STM does not compromise scalability.

Benefits of open nesting. We show that open nesting increases the envelope of concurrency and transaction sizes that can be accommodated in hardware.

The broader implications of our work are that programmers can easily make use of transactional programming abstractions to build scalable concurrent data structures without needing to devise complicated implementations using low-level synchronization primitives. Moreover, these transactional implementations can benefit from hardware acceleration on current hardware for modest transaction sizes and degrees of concurrency. We also suggest that HTM would be even more useful if its capacity were higher.

2 BACKGROUND

Researchers and implementers have explored a number ways in which transactions might be nested. A natural form of nesting for transactional constructs in a programming language is *linear nesting*, which allows a *parent* transaction to invoke a sequence of sub-operations, some of which may themselves also execute as *child* subtransactions. How these subtransactions are managed may vary, so long as the atomicity of the parent transaction is preserved. If the parent transaction aborts and its effects are discarded, then the effects of its committed children must also be discarded. Nesting is desirable when aggregating atomic operations against underlying data structures into larger transactions. For example, a transaction transferring a balance from one bank account to another needs to debit from one account while crediting the other, both operations ideally appearing to occur simultaneously, perhaps to avoid arbitrage. The debit and credit operations must themselves be implemented as atomic operations. Performing the transfer as a transaction that executes the nested debit and credit actions (in either order, it does not matter) satisfies the requirement that the balance be seen to be in one account or the other at all times. Linear nesting matches well both static nesting of program blocks in one another and the dynamic nesting patterns of calls and returns. Hence our transactionalized version of Java uses linear nesting.

Approaches to handling linear nested transactions that we consider in this dissertation include *flattening*, *closed and open nesting*, and *boosting*.

2.1 Flattening

Flattening ignores the nesting structure and runs the operations of any nested transaction as part of its parent. If a nested transaction aborts, then the entire top-level transaction also aborts. Thus, all the work of the top-level transaction must be discarded and retried.

Flattening means that no metadata for nested transactions needs to be maintained, other than a simple counter to track nesting depth—entering a nested transaction increments the counter, and committing decrements the counter. When the counter decrements to zero the top-level transaction commits its writes and they become globally visible.

Current HTM implementations, such as Intel's TSX, flatten hardware transactions. As a result, they are susceptible to failure if they run for a long time (increasing the likelihood of conflicts or interrupts) or touch a large amount of memory (exceeding capacity).

2.2 Closed Nesting

Closed nesting allows a nested transaction to abort independently of its parent. A closed nested transaction can successfully commit, in which case its reads and writes accrue to its parent. If the child aborts then its writes are discarded and the nested transaction can be retried. After some number of unsuccessful retries the parent itself may be aborted (or the parent might attempt some other action). Closed nesting sometimes avoids the need to discard the accumulated effects of a parent. On the other hand, as the write sets of a series of linear nested transactions accrue to the parent, its chances of failure due to conflict with other transactions will increase, because the write sets are larger and held longer.

Two *nested* transactions conflict as before (if they both access the same memory unit and at least one of them writes it), excepting that a child never conflicts with its ancestors. Thus, writes by children override writes of their ancestors without conflicting. Similarly, reads by children do not conflict with writes of their ancestors (but need to see the value most recently written by ancestors and previously committed descendants).

2.3 Open Nesting

Open nesting allows *further* increases in concurrency [42], by releasing concrete resources (e.g., memory reads and writes) earlier and applying conflict detection (and roll back) at a higher level of abstraction. For example, transactions that increment and decrement a shared memory location would normally conflict, since they write to the same location. But, since increment and decrement commute as abstract operations, they can be implemented correctly with open nesting. An increment (say) does: read, add-one, write. The open nested transaction would be over, its writes made globally visible, and the updated field would not be part of the parent transaction's read or write set. Instead, if the parent later aborts, it must run a *compensating* decrement to undo the logical effect of its committed open nested child.

The only difference between open and closed nesting with respect to memory accesses concerns what happens when a transaction commits. When an open nested transaction commits then its writes become permanent and *globally* visible; they do not accrue to its parent. Moreover, for each of its writes any corresponding read by its ancestors from the same location is also forgotten (so that its ancestors can no longer have conflicts on that location).

Instead of conflict detection being performed on the concrete level of memory units, when a committing open nested transaction releases its concrete reads and writes, it must typically claim some (set of) *abstract* resource(s) ("abstract locks") and provide a corresponding abstract compensation operation (e.g., the decrement in the earlier example) for use by its ancestors if they need to abort and undo the child.

If we view transaction conflicts and rollback in terms of *operations*, we can see greater similarity between closed and open nesting and highlight better the essential difference. Closed nesting works in terms of *read* and *write* operations, with the usual conflict rules on those operations. The undo of a *write* is a corresponding *write* that installs the original value of the memory unit. In the open nesting case we have a programmer-defined set of operations, with programmer-defined conflict rules and programmer-supplied rollback operations for each forward operation. So the essential difference when viewed from "outside" the transaction is the set of operations over which the transaction operates.

However, the more abstract[1] transactions provided by open nesting—which offer increased concurrency because abstract concurrency control captures the essential semantic conflict while read/write level conflict detection over-estimates conflicts—must be built from *something*, and the individual operations must still appear to execute atomically. More precisely, they must be *linearizable* [27]: they must appear to occur at a single instant of time. Transactions are one way to achieve that linearizability, so it is natural to *implement* open nesting using much the same mechanism as for closed.

Interestingly, because open nested children discard their physical reads and writes they are particularly amenable to acceleration using hardware, even when their parent runs in software. All that needs to be done is to ensure that the necessary abstract locks are acquired before the hardware open nested child commits. By storing abstract lock meta-data in a carefully-implemented (non-transactional) concurrent data structure the abstract locks can simply be acquired before entering the open nested hardware transaction (so avoiding placing the burden of managing the locks on the hardware transaction, and leaving it only to track application-level memory accesses).

2.4 Boosting

Transactional boosting [24] recognizes that *how* linearizability is achieved does not matter, and thus naturally supports an approach where existing *non-transactional* (but otherwise thread-safe (linearizable)) code is extended with transactional wrappers. For example, given a thread-safe data structure such as Java's "ConcurrentHashMap", where concurrent operations to manipulate the map are linearized using low-level non-blocking primitives, linearizability of the *composition* of sets of these operations can be achieved using the same abstract concurrency control mechanisms as for open nesting. Instead of using transactions to linearize the sub-operations (say, adding and removing from the map), transactions are used only to linearize aggregations of those sub-operations.

[1]We mean “abstract” in that conflicts don’t occur at the physical level.

For example, an aggregate operation that adds two elements to a "ConcurrentHashMap" can be linearized with respect to other operations on the map by locking the visibility of those elements until the aggregate operation completes, and providing a compensating action that removes the elements if the aggregate must be rolled back.

The advantage of transactional boosting is that it removes the need to manage low-level conflicts using transactions, which in the case of software transactional memory can have significant overhead. Instead, the underlying data structures support linearizability of their operations using other means, such as low-level hardware atomic operations. Software transaction mechanisms come into play only when it comes to aggregating these operations, capturing their resource reservations in the form of abstract locks.

2.5 Related Work

We now briefly discuss other related work. We first discuss how HTM support has evolved over the years. We then take a look at some key transaction proposals with an emphasis on hybrid transactional systems.

Ever since Herlihy and Moss [25] introduced *Transactional Memory* in 1993 there have been many proposals that exploit hardware to perform transactions. Stone et al. [55] proposed a multi-word Oklahoma Update mechanism around the same time as Herlihy and Moss. The Oklahoma Update mechanism was an extension of the Jensen et al. [31] proposal to use multiple reservation registers, whereas Herlihy and Moss proposed a transactional cache. Lie [37] and Ananian et al. [1] argued that hardware transactional memory should support unbounded transactions which led to their proposal, Unbounded Transactional Memory (UTM). Hammond et al. [22] proposed Transactional Coherence and Consistency (TCC) which was also a form of unbounded transactions but needed radical changes in hardware. Transactional Lock Removal (TLR) was proposed by Rajwar and Goodman [49] which is a form of HLE.

More recently hardware vendors have devised extensions for HTM hardware. Advanced Synchronization Facility (ASF) [10] is a hardware extension for AMD64 that introduces new instructions for specifying regions that execute speculatively. Rock [8; 9] was a multicore SPARC® processor that provided HTM instructions to begin and end speculation regions similar to ASF. There are commodity processors today that exhibit HTM capability. Intel's Transactional Synchronization Extensions (TSX) [56] and HTM support in IBM's System Z [30] are noteworthy.

2.5.1 Unbounded Transactional Memory(UTM)

Lie's hybrid TM system called Unbounded Transactional Memory [37] was one of the first Hybrid TM systems. UTM supported a form of flat nesting where each nested transaction is subsumed by its parent. Its STM design is object based and similar to that of Dynamic Software Transactional Memory (DSTM) [26]. Its hardware design was such that small transactions that fit in the cache used cache coherency protocols to detect conflicting transactions. This was the common case. Rather than limiting transaction size it allowed transaction state to overflow from the cache to main memory. UTM supported transaction sizes as large as what the virtual memory system could support and was an idealized design for HTM. However implementing it required significant changes to both the processor and the memory subsystem. So it was never fully realized. A restricted version of UTM was explored as a detailed cycle-level simulation using UVSIM [58]. It was called Large Transactional Memory (LTM). Transaction sizes in LTM could be as large as the physical memory and its duration less than a time slice. It did not allow transactions to migrate between processors. These limitations meant that LTM could be implemented by modifying the cache and processor core.

2.5.2 Kumar's Hybrid TM

Kumar et al. [35] proposed a hybrid transactional memory system that was similar to that of Lie [37]. The system starts in HTM and switches to STM when failures occur. The hardware mode detected conflicts at the granularity of a cache line while the software implementation detected conflicts at object granularity similar to DSTM [26]. The hardware design introduced a transactional buffer that recorded two versions for each line: the transactionally updated value and the current value. Two bits were also associated with each hardware thread, these indicated if the thread was executing in a transaction and if the execution was in hardware or software mode. Two bit-vectors recording reads and writes were associated with each cache line, and used by the HTM to detect conflicts. The system was evaluated on a cycle accurate, execution driven, multiprocessor simulator.

2.5.3 Virtual Transactional Memory (VTM)

Virtual Transactional Memory (VTM) was proposed by Rajwar et al. [50] to virtualize platform specific resource limits in a user transparent manner. It has a combined hardware/-software system architecture and enables users to obtain the benefits of TM without having to explicitly handle resource or scheduling limitations. VTM remapped evicted locations in virtual memory to new locations when transactions failed because of buffer overflow. If a transaction failed because of an interrupt VTM would save its state in virtual memory and resume the transaction later.

2.5.4 Hybrid Transactional Memory (HyTM)

Hybrid Transactional Memory (HyTM) [15; 35] generates separate software paths for HTM and STM with instrumentation to check the needed metadata. HyTM supported two simple back-off schemes to transition from HTM to STM in the face of failures. In the "immediate fail-over" scheme a transaction failing in HTM retries itself in STM immediately. In the "back-off" scheme, a transaction failing in HTM retries for 10 times before retrying under STM. Since the authors used transactions that were very short and with small memory footprint, the simple approach of trying HTM first for every transaction was a successful policy. Matveev and Shavit [38] describe a similar back-off policy.

2.5.5 Phased Transactional Memory (PhTM)

PhTM [36] took an alternative approach to HyTM by running transactions in phases. Under this scheme transactions cannot run in HTM and STM concurrently: it is either all HTM or all STM. This works well when all transactions succeed under HTM, but incurs major overheads if even one HTM transaction fails.

2.5.6 Hybrid NOrec

Hybrid NOrec [14] attempts to get the best out of both HyTM and PhTM by allowing concurrent hardware and software transactions without the overhead of per-access instrumentation. Its STM implementation is based on NOrec [13]. NOrec does not employ per-location metadata, but rather depends on a single global sequence lock for concurrency control. A consequence of this design is that only a single writer can commit and perform writeback at a time. Thus it scales well when its single-writer commit serialization does not represent the overall application bottleneck. Hybrid NOrec was evaluated on Rock[8; 9] and the PTLsim ASF simulator [57].

2.5.7 Deuce STM

Deuce [33] was an STM framework designed for Java that enabled transaction support without having to modify the JVM. It dynamically instruments classes at load time and uses an original “field-based” locking strategy to improve concurrency. Deuce relies on the programmer to annotate his/her code appropriately to mark transaction boundaries. The framework duplicates each method, one serves as the transactional version while the other is non-transactional.

2.5.8 Adapt STM

adaptSTM [45; 46] uses online monitoring to tune parameters that could effect STM performance. The system adapts different parameters such as write-set hash-size, hash-function, and write strategy based on runtime statistics on a per-thread basis. adaptSTM, was a word-based STM library based on a global clock and an array of combined global versions (timestamps) and locks.

3 XJ LANGUAGE

We now present details of our extensions to Java that add transactions, with open nesting, closed nesting and boosting, to the language. While closed nesting need not be associated only with classes, we connect open nesting and boosting with classes. In order to avoid deadlock internal to the transaction system, the design prevents any given static or instance field from being accessed by both closed and open nested transactions. Associating open nesting with classes also facilitates this segregation.

3.1 Atomic Actions

A block (or method) may be designated `atomic`, by writing the keyword `atomic` where the keyword `synchronized` is permitted. A block (or method) cannot be both `atomic` and `synchronized`.[1] Each execution of an `atomic` block (which includes method bodies) occurs as an *atomic (trans)action*.[2] An atomic action has three possible outcomes:

- It can *succeed*, in which case its effects are *committed*.
- It can *abort*, in which case its effects are *undone*, and the action will be *retried* from the beginning.
- It can *fail* (complete abruptly), in which case its effects are undone specially (§3.1.5) and the action is *not* retried. Action failure results from throwing of exceptions.

The *effects* of an atomic action include assignments to (shared) instance and static fields, and (unshared) local variables and formal method parameters and exception handler parameters (i.e., all declared variables), as well as the effects of nested atomic actions that it executes (see §3.2 for consideration of the case of *open* atomic actions).

[1]We may propose to remove `synchronized` entirely.
[2]We use the term *atomic action* for brevity, to refer to the execution of an atomic block/method as a transaction.

In addition to designating `atomic` methods individually, one may write `atomic` as a class modifier. This causes all methods of the class to be implicitly `atomic`. Any class that extends an `atomic` class is implicitly also `atomic`, unless the extending class is explicitly marked `openatomic` (see §3.2).

3.1.1 Effect Logging

It is helpful to consider the run-time system as (conceptually) associating with each thread a *log* of all the thread's assignments. Each record in the log indicates the variable that was assigned and the variable's previous value (§3.2.3 extends this model to include other kinds of log records). Undoing the effects of an atomic action requires processing each of the log records since the action started, from last to first, restoring each variable to its logged prior value.[3] Undoing also discards each log record after it is processed. Likewise, committing a top-level action discards that thread's log records. Committing a non-top-level action appends its log to its parent's log.

3.1.2 Concurrency Control

If a thread reads a variable while executing an atomic action, the variable is said to be a member of the action's *read set*. Likewise, if a thread writes a variable while executing an atomic action, the variable is said to be a member of the action's *write set*. An action's *accessed variable set* is the union of its read set and its write set. If the write set of an action has a non-empty intersection with the accessed variable set of another thread's action, the actions are said to *conflict*. If two concurrent actions conflict, then at least one of them must abort.

[3]Undoubtedly many optimizations are possible!

3.1.3 `Retry` Statement

The `retry` statement allows explicit programming of abort. It is useful in implementing open atomic concurrency control (§3.2.6), etc. When a thread executes a `retry` statement, the atomic action aborts immediately, and will be retried from the beginning of the action's block. Executing a `retry` statement when not in an atomic action causes a run-time error exception to be thrown.

```
RetryStatement :
  retry;
```

Syntactically, a `retry` statement can appear anywhere a `return` statement can appear.

3.1.4 `Require` Statement

The *require* statement supports *conditional* atomic actions:[4]

```
RequireStatement :
  require Expression ;
```

The `Expression` must be boolean-valued. The effect of evaluating

```
require exp ;
```

is similar to evaluating

```
if (!exp) retry ;
```

However, an implementation may be able to use knowledge of the required condition to avoid retrying if the condition's value cannot have changed.[5]

[4]We considered calling this *wait* or *await*, but its semantics are different enough from Java's current wait/notify model that we prefer to emphasize that it is different.

[5]We considered as an alternative the `watch` statement of Atomos [4], but felt that because it is so low level, it might overly constrain implementation strategies. Also, if a programmer mentions too small a watched variable set, then the program can surprisingly wait forever.

3.1.5 Exceptions

If an exception is thrown and not caught within an atomic action (i.e., the atomic action would complete abruptly), the atomic action fails, and is undone in a special way, as follows. Exception objects that are constructed and thrown, and new objects reachable from them, should not have effects related to them undone. If those effects were undone, the objects would have their fields reset to the value before any initializers were run (i.e., zero). Therefore, an implementation must not undo effects on fields of objects created since the action began. Moreover, at the time of an exception, this enables programmers easily to capture and communicate the state of previously existing objects using cloning or other copying of state into the corresponding exception object. This state will survive the failure of the enclosing action.

3.2 Open Atomic Classes

A class can be declared with the (new) modifier `openatomic`. This indicates that the open atomic instance or static *fields* of the class can be accessed only during execution of open atomic instance or static *methods* of the class.

> *Commentary:* As noted with our principles (§1.3), `openatomic` is a property of a class because all operations of the abstract data type implemented by the class need to cooperate in providing suitable abstract concurrency control and recovery.

The `openatomic` modifier is independent of the other usual class modifiers (`abstract`, `final`, etc.), and applies equally to enumerations and nested classes. Of course, a class cannot be both `atomic` and `openatomic`. Any class that extends an `openatomic` class is implicitly also `openatomic`. An `openatomic` class can extend an `atomic` class, but an `atomic` class cannot extend an `openatomic` class. We detail the reasons for this in §3.2.4.

Interfaces cannot be declared `openatomic` (which is a semantic and implementation property, not affecting signature or usage).

3.2.1 Open Atomic Fields

All `private` or `protected` instance fields of `openatomic` classes are open atomic. Only `private` static fields of `openatomic` classes are open atomic. All accesses to open atomic fields are statically guaranteed to occur during the execution of an open atomic action. All other fields are not open atomic, and a warning will be emitted at their declaration in an `openatomic` class.[6] Any field with the `final` modifier is treated as open atomic irrespective of its access modifier (this allows a `final` field to be accessed by open atomic methods and also from elsewhere, according to its access modifier).

3.2.2 Open Atomic Methods

A method is considered to be open atomic if it has at least one of the following clauses attached: `onabort`, `oncommit`, `onvalidate`, `ontopcommit`, or `locking`. (The first four are introduced in §3.2.5; `locking` clauses are described in §3.2.6.) Only an `openatomic` class can have open atomic methods. Moreover, all public or package access methods of an `openatomic` class are implicitly open atomic; they cannot be `atomic`.

Commentary: These rules are intended to prevent calls from outside the class that access open atomic *instance* fields other than during execution of an open atomic method on that *instance*, and likewise to prevent access to open atomic *static* fields other than during execution of an open atomic *static or instance* method. We assume that open atomic *instance* methods that directly or indirectly access open atomic *static* fields provide suitable *class*-level concurrency control and recovery.

Private or protected methods of `openatomic` classes can still be non-atomic. They can also be `atomic`, allowing a method that atomically composes invocations of two or more open atomic methods, for example.

[6]Because `public` and `package` access fields can be accessed directly from outside of the class, we cannot restrict them to be accessed only during execution of open atomic methods. Similarly, `protected` static fields can be accessed directly from subclasses, so we cannot restrict their access to occur during execution of open atomic methods.

3.2.3 Open Atomic Method Execution

An open atomic method always executes as an atomic action. However, if it completes successfully (commits), its writes are made permanent (globally visible), and its log is discarded. Moreover, if the open action is also nested then it has the following effects on its parent's log:

- Its handler clauses (`onabort`, `oncommit`, `onvalidate`, and `ontopcommit`, if exist) *take effect* (are logged). Clauses in effect may later be executed, under certain conditions.

- It acquires abstract locks, as described in its `locking` clauses (if any), which are logged.

Discarding its log means that any clauses in effect from open atomic actions on other instances or classes, committed during this open atomic action, become no longer in effect. Discarding its log also means releasing locks held from such actions.

Because the open atomic public/package methods of an `openatomic` class are its only external entry points, each of which begins an open atomic action on entry, all methods of an `openatomic` class are guaranteed to execute in the dynamic context of an open atomic action, or nested within one. There are occasions when one open atomic method may *internally* call another open atomic method in the same class (or superclass), in which case their effects are *aggregated*, merging the open atomic callee into the caller's action. This avoids the need to duplicate internal subtransaction handlers in their parent's handlers.

For example, if a linked list class has open atomic `add` and `remove` methods, one might write an open atomic `move` method to move an item from its current position to the end of the list. If `move` is written as `remove` followed by `add`, then the `onabort` actions for both `move` and `add` accrue to `move`, instead of being discarded when `move` commits. Otherwise, one would be forced to duplicate them in the `onabort` clause for `move`.

Commentary: One might implement aggregation as follows. For each open atomic method `m` create a corresponding "internal method" `mInternal` having the same signature and body, but not open atomic. Rewrite internal calls of `m` to call `mInternal`.

When an open atomic method completes successfully, its open atomic clauses and locks, some of which may come from aggregated calls, are *logged* at that time to its parent. Thus the log described in §3.1.1 also contains records for `onabort`, `oncommit`, `onvalidate`, `ontopcommit`, and `locking` clauses.

Undoing an atomic action (because of abort or failure), processes its portion of the log in reverse order (as in §3.1.1). Processing ignores records such as `oncommit`, `onvalidate`, and `ontopcommit`. It also ignores records corresponding to `locking` clauses (these are released as described in §3.2.6). When undoing encounters an `onabort` record, it executes the corresponding `onabort` block as an open atomic action. Notice that undoing of writes and execution of `onabort` clauses are interleaved (but not concurrent): all occur in reverse log order. The processed log records are discarded, as described in §3.1.1. Finally, control resumes at the beginning of the aborted action, if it is to be retried, or the exception causing failure is propagated.

Committing an atomic action processes its portion of the log in forward order from the beginning to the end. Processing first runs the `onvalidate` records to ensure the transaction is in a state that can be committed. It then processes the `oncommit` records. If the committing action is a top level transaction it then processes the `ontopcommit` records. Processing these handler records causes their corresponding clauses to be executed as open atomic actions. Log records for writes, and `onabort` and `locking` records, are ignored when committing. Committing then discards the processed log records and releases all of the committing action's abstract locks (see §3.2.6). Control then continues normally.

3.2.4 Inheritance, Overriding, and Nesting

An `openatomic` class can extend a class having public/package `atomic` methods, but inheriting those methods without overriding them in the `openatomic` class is dangerous because it allows accessing fields of the open atomic instances in both open and closed execution modes. Mixing access modes in this way can lead to deadlocks [42].

To avoid this, we can either require that all inherited `atomic` methods be explicitly overridden with open atomic methods in the `openatomic` class, or implicitly "copy down" the inherited method as an open atomic method. The latter may save some typing by the programmer, but the former has the advantage of forcing her to think through the abstract locking protocol for all the open atomic methods of the `openatomic` subclass. Our inclination is towards forcing the programmer to provide explicit overrides. Invoking the `atomic` superclass method with a `super` call from the body of the overriding open atomic method (or elsewhere in the subclass) is always safe, because instance field accesses will always occur in the context of an open atomic action.[7]

Conversely, an `atomic` class cannot extend an `openatomic` class. Otherwise, calls using `super` would enable the subclass to access fields in both open and closed modes.[8]

Similarly, a nested class (either static or non-static), which can directly manipulate the open atomic fields of its outer class or instance, is implicitly `openatomic` if its outer class is `openatomic`. This ensures that external entry points via the nested class also preserve the open atomic nature of the enclosing class's open atomic fields.

One additional piece of mechanism is necessary to ensure proper handling of open atomic fields. It is possible for an open atomic method on instance o to call methods on some other objects, resulting in a call chain that comes back to calling a method on o. Unlike aggregation to construct larger open atomic actions from smaller ones `operating on the same object`, where the call chain does not leave the scope of the instance, in this case the call chain is *re-entrant* after leaving the instance. In such cases, the re-entrant open action cannot safely release its physical updates, since the outer open action on that object is still active. Thus, we also formulate an additional run-time restriction, as follows. For any given object accessed in an open atomic way, indirect (non-aggregating) re-entrant

[7]It may not be *correct*, however, unless the overriding method adds suitable `locking` and `onabort` clauses, etc.

[8]Alternatively one could have it mean something like "copy down all the methods, removing all their open atomic clauses".

calls to open atomic actions run instead as closed atomic. The requirement is analogous to the tracking of re-entrant nesting depth for Java `synchronized` blocks/methods, where the lock is released only when exiting the outermost lock level.

3.2.5 Open Atomic Method Suffix Clauses

We now give the syntax for the handler clauses that may be attached to the end of an open atomic method, namely `onabort`, `oncommit`, `onvalidate`, and `ontopcommit` clauses. A given method may have at most one of each kind of clause attached. Moreover, because the handlers may wish to use values computed at the beginning of the action, an optional list of local variable declarations can be evaluated before the method body proper. These pre-declarations (`PreDecls`) evaluate at the same level as the method body, in the scope of the formal parameters, and are delimited syntactically by square brackets `[]`. The variables they declare are in scope for both the method body and the handler clauses.

```
MethodDeclarator:
  Identifier ([FormalParameterList]) [PreDecls]
PreDecls:
  [{LocalVariableDeclarationStatement}]
MethodBody:
  Block {OpenAtomicClause}
  {OpenAtomicClause} ;
OpenAtomicClause:
  onabort     Block
  oncommit    Block
  onvalidate  Block
  ontopcommit Block
```

Supporting these constructs, and supporting use of method parameter values and pre-declarations, requires generating code that saves the necessary values and makes them available to the handler clauses if and when they run.

```
public interface LockTable
  <LT extends LockTable<LT>> {
    public void acquireLock
      (LockShape lockShape, LockMode mode,
       TxnDescriptor desc) throws LockConflictException;
    public void releaseLock(Lock lock);
}

public interface Lock
  <S extends LockShape, M extends LockMode> {
    public S getLockShape();
    public M getLockMode();
    public TxnDescriptor getTxnDescriptor();
    public LockTable getLockTable();
}

public interface LockSpace
  <M extends LockMode<M>, LS extends LockSpace<M,LS>>
  extends LockTable<LS> {}

public interface LockShape
  <M extends LockMode<M>, LS extends LockShape<M,LS>>{}
```

Listing 3.1: Lock tables, spaces, shapes, and modes

3.2.6 Open Atomic Method `Locking` Clause

We provide a framework for users to construct their own abstract locking protocols, along with several pre-defined abstract lock libraries. The locking framework relies on the notions of *locks*, *lock spaces*, *lock shapes*, and *lock modes*. The "locking" clause of an open atomic method requests locks of particular shapes in particular modes from lock spaces. The type signatures of these are illustrated in Listing 3.1. (The metaphor here is of possibly overlapping geometric shapes within some space. A shape indicates *what* is being locked, while a lock *mode* describe *how* it is being accessed.)

An instance of an open atomic class will typically have some number of lock tables in which to record abstract locks held by active transactions against the *abstract* state of the instance. Lock tables record locks and the mode in which they are held, along with the transaction holding the lock. Locks come in multiple shapes, as defined by a lock space, allowing a single lock to cover a range of locked values.

An open atomic method invocation can try to obtain one or more abstract locks. These are specified via `locking` clauses associated with the method, and `return` or `throw` statements in its body. An abstract lock needs to:

1. indicate the lock table instance in which to request the abstract lock;
2. indicate the specific lock *shape* requested (and any parameters needed for that shape) within the table's lock space; and
3. indicate the specific lock mode instance to use.

As an example, consider the design of an open atomic class "Ordered""Set<T>" implementing "java.util.Sorted""Set<T>". A suitable lock space for an ordered set is the one-dimensional set of all possible "T" instances, having a total order ("OneD""Space"). Within this space one can imagine a number of lock shapes:

Point(x): lock a single "point" object, associated with a particular "T" instance x, which mathematically could be considered the range $[x,x]$;

GT(x): lock upward "rays" starting at x, meaning $(x,\infty]$;

LT(x): lock downward "rays" starting at x, meaning $[-\infty,x)$;

Range(x,y): lock ranges defined on values x and y where $x \leq y$ in the total order, meaning (x,y), etc.

```
openatomic class OrderedSet<T> ... {
  private final
    LockSpace<SXMode,OneDSpace<SXMode,T>>
    eltSpace;
  private final
    LockSpace
```

```
enum SXMode implements LockMode<SXMode> {
  SHARED {
    public boolean conflictsWith(SXMode other) {
      return other != SHARED;
    }
  },
  EXCLUSIVE { public boolean conflictsWith(SXMode other) {
    return true;
    }
  },
}
```

Listing 3.2: Shared/eXclusive lock modes

```
enum PCMode implements LockMode<PCMode> {
  PIN {
    public boolean conflictsWith(PCMode other) {
      return other != PIN;
      }
    },
  CHANGE {
    public boolean conflictsWith(PCMode other) {
      return other != CHANGE;
    }
  },
}
```

Listing 3.3: Pin/Change lock modes

```
      <PCMode, UnitSpace<PCMode,OrderedSet<T>>

    setSpace;

  public boolean add(T elt) locking

    (eltSpace : point(elt) : SXMode.EXCLUSIVE),

    (setSpace : get() : PCMode.CHANGE) ...

  public boolean remove(T elt) locking

    (eltSpace : point(elt) : SXMode.EXCLUSIVE),

    (setSpace : get() : PCMode.CHANGE) ...

  public int size()
```

```
    locking (setSpace : get() : PCMode.PIN) ...
  public boolean contains(T elt)
    locking (eltSpace : point(elt) : SXMode.SHARED) ...
  ...
}
```

Listing 3.4: `OrderedSet` lock tables

We will use two lock mode classes here, "SXMode", shown in Listing 3.2, and "PCMode", shown in Listing 3.3. "SXMode" provides "SHARED" and "EXCLUSIVE" locks (also often called read/write locks). "SHARED" and "EXCLUSIVE" modes are used concerning the presence/absence of individual elements of a set. For the set as a whole, we can *pin* the state of the set using "PIN" mode, or indicate some *change* to the set using "CHANGE" mode: operations like "size" would use "PIN" mode, and "add"/"remove" operations would use "CHANGE" mode on the set (plus "EXCLUSIVE" mode on individual elements). Note that "CHANGE" conflicts with "PIN" but not with "CHANGE". The two mode classes "SXMode" and "PCMode" are strictly different.

Lock modes are naturally implemented using Java "enum" classes that implement the "LockMode" interface.

An "OrderedSet<T>" might then have two lock tables, one for the set of individual elements and one for gross statistics (current size, total number of insertions/deletions, etc.) about the set as a whole, as in Listing 3.4. In this example, "UnitSpace" is a space that allows locking just one object, in this case the set as a whole.

In addition to the suffix clauses, an open atomic method may acquire abstract locks before it can complete successfully. The `locking` clause is attached to the method's header, revising the syntax of *MethodDeclaration*:

```
MethodDeclaration:
  MethodHeader [LockingClause] MethodBody
LockingClause:
  locking [+]( LockExpressions )
```

LockExpressions :

  LockExpression {, *LockExpression* }

LockExpression :

  LockTableExp : *LockShapeExp* : *LockModeExp*

LockTableExp :

  Expression

LockShapeExp :

  Expression

LockModeExp :

  Expression

A locking clause is syntactic sugar for acquiring a lock from a lock table.

The *LockTableExp* must have a type that implements the `LockTable` interface; it indicates the lock table in which to request the abstract lock denoted by the locking clause. A `LockTable` encapsulates a `LockSpace` instance and a `LockMode` type. These are defined in a standard library as shown in Listing 3.1. An open atomic class will typically have one or more `LockTable` instances for representing abstract locks held on itself or its instances. The *LockShapeExp* must return a `Lock`, by invoking the indicated method on the lock table `LockSpace`, itself obtained using `getSpace()`. It indicates the specific shape requested (and any parameters needed for that shape) within the table's lock space. The *LockModeExp* must be of a type that implements the `LockMode` interface; it indicates the mode in which to acquire the lock. A `locking` clause has the same scoping behavior as the suffix clauses.

An overriding method inherits the overridden method's `locking` clause. If an overriding method supplies its *own* `locking` clause, then the overridden clause is not inherited. If a method needs to *extend* an inherited `locking` clause, it can use the optional + sign with its `locking` clause.

At the time an open atomic method execution accumulates locks, one evaluates each *LockExpression* in turn, in textual order. To evaluate a *LockExpression*, one first obtains the `LockSpace` from the `LockTable`. One then calls the method described by the *LockShapeExp*; this results in a `Lock` type. The next step is to attempt acquiring the abstract lock. This is done by calling the `add` method on the `LockTable` instance specified by the *LockTableExp*. If the `add` call completes successfully, then we say that the current transaction *holds* a lock on the specified object in the specified mode. The call may fail due to lock conflict. In this case the current transaction aborts and will be retried.

When a transaction completes (successfully or unsuccessfully) and *releases* its locks, it no longer holds them.[9]

3.2.7 Acquiring Locks at `Return` or `Throw`

Sometimes, throwing an exception indicates something about an object's state. For example, calling `remove()` on an empty `Queue` throws `NoSuchElementException`. Arguably, this should lock the fact that the queue is empty. However, our interpretation of exceptions as causing abort prevents `remove()` from acquiring such an abstract lock on the queue's state. Hence, we allow one to attach a `locking` clause to a `throw` statement:

```
ThrowStatement :
  throw Expression [LockingClause] ;
```

The indicated locks are acquired as the exception is thrown, and are logged as part of the containing action. If execution is not within an atomic action (open or not), the `locking` clause has no effect.

Similarly one can have a `locking` clause attached to a `return` statement and the locks are acquired as the result is returned and logged as part of the containing action:

```
ReturnStatement :
  return [Expression] [LockingClause] ;
```

A *LockingClause* attached to the *MethodDeclaration* is *inherited* by all `return` and `throw` statements by default. A `return` or `throw` statement may choose to override the inherited *LockingClause* by providing its own. If it wants to extend the inherited *LockingClause* it must use the optional + sign with its *LockingClause*.

[9] Since release might be implemented in batch in a variety of ways, we do not specify the interface here. Since each lock is associated with a given transaction, and is held until the transaction completes, one always releases all of a transaction's locks at the same time.

3.2.8 Open Atomic Concurrency Control

To define open atomic action concurrency control we introduce a conceptual device we call the *augmented log*. In addition to recording writes and open atomic clauses, the augmented log records reads of shared variables. An action's current read set is those variables that have a read record in the action's log, and its current write set is those variables that have an assignment record in the action's log. In the presence of open atomic actions, read and write sets can shrink as well as grow, as nested open atomic action commit and discard their related portion of the log. Beyond that, conflict is as in §3.1.2, with the addition of explicit locking specified in `locking` clauses and associated conflicts. (Notice that in this log-based view of concurrency control, the locks that an action holds are exactly those recorded in its log.)

3.2.9 Open Atomic Actions and `New`

When an atomic action aborts, what happens to objects it allocated? In the absence of open atomic actions, it is clear that no other action can have seen, or will see, the newly allocated objects, so there is no issue. However, in the presence of open atomic actions, an open atomic action can publish to a globally accessible variable a reference to an object allocated in a containing action. If the containing action aborts, and the published reference remains, to what does the reference refer? (The situation is similar to abrupt completion of constructors as discussed in the Java Language Specification (Sections 12.4 and 12.5 in the Third Edition).) We require that the compiler and run-time system guarantee that the reference refers to a type-safe instance (of the class indicated in the `new` expression). However, the instance may be partially or completely unconstructed, i.e., fields (including `final` fields) may have their default initial values. In other words, the situation may be as if the constructor has not yet run.

It is helpful if we consider instance creation to consist of *allocation* followed by *initialization* (constructor execution), as occurs in the Java Virtual Machine. We require that allocation be effectively an open atomic action. Constructor execution then proceeds with a type-safe instance of the class being allocated, each of its fields having the default value for their type. Thus, if a constructor aborts, it unwinds the instance to this default state. We observe that, as per the Java Language Specification, it is not a good idea to publish a reference before the referent is fully constructed.[10]

3.2.10 Concerning `Volatile` and `Synchronized`

Given the power of atomic actions, and open atomic actions in particular, there seems little additional value to `volatile` fields when used for synchronization. When used for applications such as access to memory-mapped I/O device registers, in the presence of atomic actions `volatile` fields may best be used within `oncommit` clauses. The same might be said concerning invocations of library routines and operating system calls.

Concerning Java `synchronized` blocks and methods, we believe that they, along with wait and notify support, can be implemented using open atomic actions in stylized ways. This would replicate their semantics faithfully. The same field should not be accessed in both atomic and synchronized code, since atomic code's undo, retry, and `oncommit` are somewhat unpredictable as to whether and when they occur.

In the long run, code using `synchronized` could be converted to either `atomic` or `openatomic`. We note that `openatomic` can be used to build any ordering and signaling mechanism desired.

3.3 Boosting

The framework for abstract locks extends naturally to boosting. In place of "openatomic" we allow a class to be declared "boostedatomic". This presumes that the implementations of its methods are inherently linearizable, such as by being implemented using non-blocking hardware primitives instead of executing as open nested transactions.

[10]It may also be useful to view constructors as being open atomic, with no `onabort` or `locking` clause, though adjustment may need to be made for their effects on other objects and on any static fields.

4 XJ FRAMEWORK

Our implementation comprises of four main components: (1) a compiler front-end based on OpenJDK's "javac" which we call the XJ compiler, (2) a minimally modified version of OpenJDK to support hardware transactional memory using TSX, (3) a Java agent for load-time bytecode rewriting to inject transaction support, and (4) a run-time library to manage the dynamics of transactions and abstract locking.

4.1 STM Implementation

Our implementation approach is similar to that of the McRT software transactional memory (STM) system [52]. McRT associates with each object (or word, the granularity being determined on a per-type basis) a *transaction record*. This record contains either a *version number* (for an object/word that does not have uncommitted writes) or a (pointer to) the *transaction descriptor* of the writing transaction. A transaction (atomic action) accumulates two lists of transaction records, one for items it reads (and the version number seen) and one for items it writes (including the old version number and the old value). It updates fields in place. When a transaction desires to commit, it must first *validate* its read set: each item must either contain a version number that is equal to what the read set recorded, or must point to the descriptor of the committing transaction (i.e., be later written by this transaction).

In the presence of nesting, open atomic actions commit by validating reads and installing new version numbers for written items. Commits of non-open atomic actions simply append their read and write set lists to those of the containing action, first updating written item transaction descriptors to refer to the parent (or we can introduce an additional level of indirection). They need not validate read sets, since the read sets need to be validated upon commit of an open atomic or top-level ancestor anyway. (Validating on nested action commit might detect conflicts earlier, but is extra work for successful transactions.)

We need an additional mechanism to group write entries so that appropriate batches of them are undone before invoking `onabort` clauses when undoing. This can be done by starting a new (closed) nested action after the commit of an open atomic action.

Our STM library's API is designed to support a range of possible STM implementations. Transactions read and write fields via accessor functions. We can change the code we generate for the accessors in order to deploy different strategies. Further, any given transaction must "open" an object before accessing it. An object may be opened for reading only, or for writing (and reading), and may be upgraded from reading to writing. Accessing an open for reading (writing) requires having the object open for reading (writing). Thus the "open" functions and the accessors are "hooks" that can be used to create almost any policy. The current prototype perform concurrency control on whole scalar objects and on chunks of arrays. Further, its atomicity strategy is to update in place, saving previous values in a write log, and to undo when necessary.

4.2 Hardware Transactional Memory

A number of CPU models now support one flavor or another of transactional memory in hardware [14; 16; 19; 30; 51; 56]. Notable is Intel's TSX feature, of which RTM, restricted transactional memory, is a part. RTM is included in recent models of the Haswell line of processors. How does this fit with our collection of kinds of nested transactions? What RTM offers is flat transactions, executed in a best effort fashion in hardware. An RTM transaction can fail for transient causes, such as conflicts with accesses by another hardware thread, or may always fail. This is because RTM keeps all pending updates in the first level cache, to make discarding them easy in case of transaction abort. This means that if any line in the set associative cache overflows a transactional entry, the transaction will abort. Certainly a transaction cannot access more data than fit in the first level cache.

While for some applications it would be painful to provide a software fall-back for when a hardware transaction fails, in XJ we already have a complete software TM implementation. Therefore, it is easy for XJ to attempt a transaction in hardware and fall back to software as necessary.

For the most part, hardware TM implementations of atomic code in XJ do not need to do as much bookkeeping as STM. Still, in order to work properly with concurrent *software* transactions accessing the same objects, HTM code must examine the version/lock field for objects it accesses, and increment the version number for those it writes. It is easy to see how the version/lock field appropriately connects the success/failure of both hardware and software transactions into what is called a *hybrid* scheme. In the case of open nesting, it is not necessary to acquire an abstract lock and then discard it. However, it *is* necessary to check whether the lock *could* be acquired. In any case HTM adds more atomic code variants to the zoo.

In the face of nesting, HTM imposes some restrictions as to how certain transactions can be nested. Because the current hardware implements flat transactions only, it does not make sense to have a software transaction running inside of a hardware one. Thus, once a transaction starts executing under HTM it stays in HTM. With closed nesting you need to aggregate the log on commit, but running HTM inside of an STM transaction does not allow this. Thus XJ does not allow running a transaction under HTM when a parent transaction is running under STM. On the other hand if the parent is running open nested, then the child transaction can run under either HTM or STM, because abstract locks and undo semantics can be used to undo any changes.

4.3 Example: An Open Atomic `Map`

We illustrate using the features of the XJ language by presenting an openatomic implementation of the `Map` interface. The mechanisms used for open nesting in general subsumes those used for closed nesting and boosting, hence our example focuses on an open nested implementation. Listing 4.1 shows how an open atomic implementation of the `Map` interface can be defined as a concurrency-safe wrapper for unsynchronized `Map` implementations: `OpenMap` is declared as an `opanatomic` class implementing the `Map` interface, and permits safe concurrent access to the wrapped map, with `get`, `put`, `remove`, and `size` operations defined as `openatomic` methods.

Generally, `onabort` handlers are needed only for methods that mutate the abstract state of the map. The `put` operation returns the previous value associated with the given key in the map, or `null` if there was none. Thus, the `onabort` handler for `put` must either revert the map to contain that previous association if there was one, or simply remove the new association. Likewise, `remove` returns the previous value if any, so its `onabort` handler must restore that previous association.

```
public openatomic class OpenMap<K, V> implements Map<K, V> {
  private final Map<K, V> map;
  private LockSpace<SXMode, PointSpace<SXMode, K>> keySpace
      = new PointSpace<SXMode, K>();
  private LockSpace<PCMode, UnitSpace<PCMode, OpenMap<K, V>>> mapSpace
      = new UnitSpace<PCMode, OpenMap<K, V>>();

  public OpenMap(Map<K, V> map) {
    this.map = map;
  }

  public V get(Object key)
  locking (keySpace:point((K)key):SXMode.SHARED) {
    return map.get(key);
  }

  public V put(K key, V val) [ V result ]
  locking ( keySpace : point(key) : SXMode.EXCLUSIVE,
    mapSpace : get() : PCMode.CHANGE) {
      result = map.put(key, val);
      return result;
    }
    onabort {
      if (result == null)
        map.remove(key);
      else
        map.put(key, result);
    }

    public V remove(Object key) [ K k = (K)key, V result ]
    locking ( keySpace : point((K)key) : SXMode.EXCLUSIVE,
      mapSpace : get(): PCMode.CHANGE) {
        result = map.remove(key);
        return result;
    }
    onabort {
      if (result != null)
        map.put(k, result);
    }

    public int size()
    locking ( mapSpace : get(): PCMode.PIN) {
        return map.size();
    }

    // ... other methods of the Map interface
}
```

Listing 4.1: `OpenMap` class

The example uses the lock modes `SHARED` and `EXCLUSIVE` of the "SXMode" class shown in Listing 3.2, and the `PIN` and `CHANGE` modes of the "PCMode" class shown in Listing 3.3, with compatibility defined by their `conflictsWith` methods. `SHARED` locks are compatible with each other since multiple readers can operate on the same data item (i.e., `key`) at the same time. On the other hand, one cannot write a data item while it still has readers, nor read a data item while it has a writer. `CHANGE` locks reveal, at a coarser granularity, that some writer is modifying some portion of a larger data item—in this case the map itself. Thus, to `put`/`remove` an association for some `key` in the map requires a `CHANGE` lock on the map as a whole. Two requests to `put`/`remove` an association for different keys do not conflict. However, to `put`/`remove` an association for any given key does conflict with requests that read the state of the map as a whole, such as the `size` operation. The necessary constraints are recorded for `put`/`remove` by acquiring an `EXCLUSIVE` lock on the `key`, to prevent others from changing that association, along with a `CHANGE` lock on the map to prevent others needing shared mode access to the whole map (such as `size` requires).

4.4 XJ Compiler

Our implementation of the XJ compiler is based on version 1.7.0-ea-b19 of OpenJDK's `javac`. This has been extended to accept the new XJ syntax and generate compliant Java bytecode that will run on any standard Java virtual machine (though transaction support comes only when combined with the XJ run-time rewriter and XJ run-time library, the generated bytecode is transaction protocol agnostic). We modified the parser to accept the new syntax, the annotation processor to statically check the new constructs, the abstract syntax tree (AST) to represent handlers and lock expressions, and the lowering phase to transform the high-level XJ constructs into a standard Java AST. We had no need to modify the bytecode generation parts of `javac`.

We focus our explanation of the compile-time transformations on those needed for open nested methods, which subsume those for closed atomic methods / blocks and boosted methods, illustrated for the `remove` method of the `OpenMap` example shown in Listing 4.1. The XJ compiler produces Java bytecode for this method equivalent to the Java source shown in Listing 4.2, as follows.

- The PreDecls in a MethodDeclarator transform into local variable declarations in the method body, allowing the capture of state at the beginning of the open nested action, as seen at line 3.

- Lock expressions can be inherited by overriding methods. To facilitate this we transform lock expressions into protected methods of a class and invoke the method at the point the lock needs to be acquired (line 11).

- Suffix clauses are encapsulated as anonymous instances of inner classes that capture their unbound variables from the enclosing scope as final variable declarations as described in detail below.

```
1  public V remove(Object key) {
2    TxnDescriptor _$current_desc = null;
3    K k = (K)key;
4    V result = null;
5    boolean _$succeed = true;
6    while (true) {
7      try {
8        _$succeed = true;
9        TxnDescriptor.beginOpen(_$current_desc);
10       try {
11         this.remove_$locking((K)key,_$current_desc);
12         result = map.remove(key);
13         return result;
14       } catch (TxnException ex) {
15         TxnDescriptor.abortOpen(_$current_desc);
16         _$succeed = false;
17         continue;
18       } catch (Error ex) {
19         TxnDescriptor.abortOpen(_$current_desc);
20         _$succeed = false;
```

```
        throw ex;
      }
    } finally {
      if (_$succeed) try {
        final K _$k = k;
        _$current_desc.getOpenLog()
            .logHandler(new TxnHandler() {
          public void _$abort() {
            if (result != null) {
              map.put(_$k, result);
            }
          }
        }, TxnHandler.ON_ABORT_HANDLER);
        TxnDescriptor.commitOpen(_$current_desc);
      } catch (TxnException ex) {
        TxnDescriptor.abortOpen(_$current_desc);
        continue;
      }
    }
  }
}

protected void remove_$locking (K key, TxnDescriptor _$current_desc) {
  LockShape shape = ((PointSpace)keySpace).point(key, _$current_desc);
  keySpace.acquireLock(shape, SXMode.EXCLUSIVE, _$current_desc);

  shape = ((UnitSpace)mapSpace).get(_$current_desc);
  mapSpace.acquireLock(shape, PCMode.CHANGE, _$current_desc);
}
```

Listing 4.2: Transformed `remove` method of `OpenMap`

4.4.1 Handlers on Open Atomic Methods

If an open atomic method runs to completion then its handlers need to be logged. In the case that it fails it must retry from the beginning. To allow retry we wrap the method body in a `try/finally` block (line 7 of Listing 4.2) nested within an infinite loop (line 6). The outer `try/finally` block is used to detect the successful completion of the method. In the case that it does complete successfully we then create a new instance of a `TxnHandler` class overriding the corresponding method defined in the handler, and log the handler (line 26). We then commit this open atomic transaction. If a `TxnException` occurs while trying to log the handler, we abort the transaction and retry it from the beginning. Inside of the main `try/finally` block is another nested `try/catch` block. This is used to run the corresponding method body (line 12). Prior to running the method body we create a new nested transaction (line 9). If the collapsed method throws a `TxnException` we abort the transaction and retry the transactional method. In the case where an `Error` is thrown we abort the transaction and throw the `Error`. In either of these cases we avoid logging the handlers; they are logged only when the method completes successfully.

It is possible for a constructor of an open atomic class to have handlers associated with it. The transformation described above cannot be applied to constructors directly because the first statement in a constructor should be a call to a superclass constructor or another constructor in the current class. To get around this issue we use a two phase transformation for constructors of open atomic classes. The first phase is done by the XJ compiler while the bytecode instrumenter completes the second phase. The XJ compiler leaves the `super` or `this` call as it is in the constructor (even if it has complex expressions as arguments to the other constructor) and moves the rest of the statements to a `get_$init` method. A call to this `get_$init` method is added to the constructor. The transformation done in the second phase is explained in Section 4.6.3.

```
public final class Unsafe {
...
   public static int beginHWTxn() { return 0; }
   public static void endHWTxn() {}
   public static void abortHWTxn(int flag) {}
...
}
```

Listing 4.3: HTM methods added to sun.misc.Unsafe.java

4.5 OpenJDK Modifications

In order to make use of the new TSX instructions to support HTM we need a modified Java virtual machine capable of injecting them into compiled code. We augmented version 7u40-b23-2013-08-26 of OpenJDK. The TSX specification provides two different interfaces to programmers. While both interfaces make use of the underlying TM hardware, their purpose is quite different. The Hardware Lock Elision (HLE) interface is used to implement hardware lock elision techniques while the Restricted Transactional Memory (RTM) interface resemble classic TM proposals. We use RTM since it is more amenable to implementing HTM.

We modify the non-standard `sun.misc.Unsafe` class of OpenJDK as shown in Listing 4.3 to provide methods that begin, end, or abort a hardware transaction. We do not provide any concrete implementations of these methods here, but instead provide their implementations via HotSpot compiler and interpreter intrinsics [32]. We use `sun.misc.Unsafe` as a mere interface to communicate between the user code and the HotSpot compilers (both C1 and C2) and interpreter. Providing intrinsic implementations of these methods avoids the overhead of calling them as native code routines. These intrinsics were the only extensions we made to HotSpot.

The `beginHWTxn` method uses the new `XBEGIN` instruction. If the transaction completes successfully it returns -1; in the failure case it returns the corresponding error code stored in the `EAX` register. This code can be used to diagnose the reason for the hardware transaction's failure. The `endHWTxn` method uses the new `XEND` instruction, which indicates the end (commit point) of a hardware transaction. The `abortHWTxn` method can be called if the transaction needs to be aborted explicitly. This method uses the `XABORT` instruction and takes an `int` flag as an argument, which fills in part of the `XBEGIN` result code in `EAX`, allowing the caller of `abortHWTxn` to convey a few bits of information outside the aborting transaction.

In our initial experiments, many simple hardware transactions surprisingly failed due to conflicts even when there were no writes involved. We diagnosed this issue using the Intel Software Development Emulator (SDE) and found the conflicts to occur on accesses to a bookkeeping field of the `Node` class in the tree data structure manipulated by our benchmark. These conflicts turned out to be due to false sharing because TSX operates at the granularity of a cache line. Java 8 introduced the `@Contended` annotation to be used to prevent such false sharing. We back-ported this feature to Java 7 and added suitable `@Contended` annotations to the benchmark code.

4.6 Bytecode Rewriter

To add transaction support to classes we adopt an approach similar to that of the work in transparent distribution for Java [39], allowing mediation of all accesses to static and instance fields, as well as elements of arrays. The transactional machinery needed by objects (the lock word, etc.) reside in instances of `TxnObject`. Ideally, all objects that are going to be read from or written to inside a transaction extend this class. Also reads and writes inside transactional methods and transactional blocks need to be logged. We accomplish this by instrumenting classes at run time. The instrumentation that needs to be performed on a class depends on the classification of the class. We divide classes into two categories, *direct* classes and *wrapped* classes. Direct classes are ones that can be transformed to inherit from `TxnObject` and on which our rewrites can be performed directly. Figure 4.1 shows the manner in which direct classes are transformed.

There are a few classes in the JVM that cannot be rewritten directly in this ideal manner. The JVM has intimate knowledge of these classes; e.g., the offsets of fields in these classes

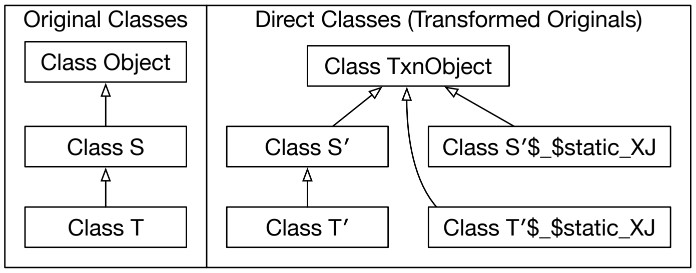


Figure 4.1.: Transformation of direct classes

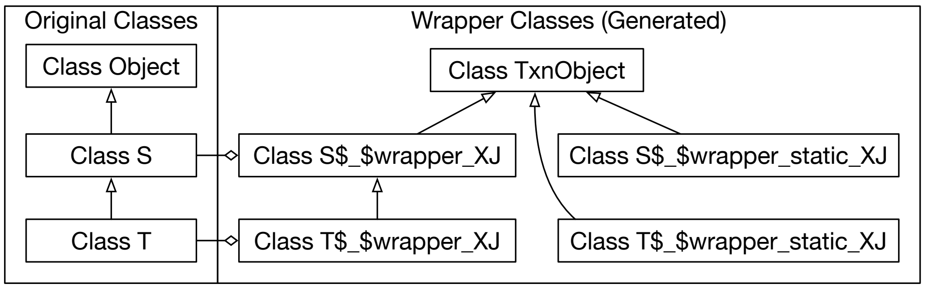


Figure 4.2.: Transformation of wrapped classes

are hard coded into the JVM (`java.lang.ref.SoftReference` in Oracle's Hotspot JVM is an example), thus they cannot extend `TxnObject`. In order to get the transactional machinery into these classes we *wrap* them. We also wrap all classes that have native methods.[1] The manner in which wrapped classes are transformed is shown in Figure 4.2.

We preprocess all classes used by the application prior to running it. Preprocessing helps us classify classes beforehand. The process used is similar to that of McGachey et al. [39].

4.6.1 Statics

Object locking in XJ is done via a lock field in `TxnObject`. This mechanism does not work for static fields, since they are not part of an object. In order to use the same transactional machinery on static fields we move the static fields and static methods of a class to a generated class, where they become instance fields and instance methods. We

[1]This is safe, but it may not always be necessary, depending on how JNI libraries are coded.

also generate get/set methods for these moved fields. We guarantee that there is only one instance of this generated class, which we call the *static singleton* of the original class. The static singleton is initialized via the static initializer of the generated class. When a static singleton is initialized it also initializes its superclass, which would be the static singleton of the original class's superclass. Each static singleton class has a static `get_$singleton` method to get the single instance. The instrumenter rewrites `getstatic` and `putstatic` bytecodes to first obtain the corresponding singleton for the field and then invoke the appropriate get/set method on it. `invokestatic` is also rewritten such that the invocation is on the static singleton instance.

4.6.2 Arrays

We generate special "array classes" for array types. This helps us get the transactional machinery into arrays. Arrays do not use the same locking mechanism used by scalar objects. Having a single lock word for the whole array would not perform well. Instead, we allow customizing of the lock scheme used on arrays, having a lock for each element or a lock for each portion of the array. The `TxnArray` interface defines the API for obtaining locks on an array. The XJ run-time library provides wrappers for each primitive array type and for the object array type. Each generated array class extends one of these wrapper classes, enabling it to gain access to the transaction machinery. The structure of the generated array classes is similar to that of McGachey et al. [39] for arrays. Figure 4.3 shows the transformation for array types.

4.6.3 Object Creation

As mentioned before, constructors go through a two-phase transformation. The second transformation is performed by the bytecode rewriter. The purpose of this transformation is to move all the code from inside the constructor to the `get_$init` method. We do this by adding a dummy constructor to each class. The dummy constructor is used purely for object creation. This enables us to create an empty object for a given class. We then

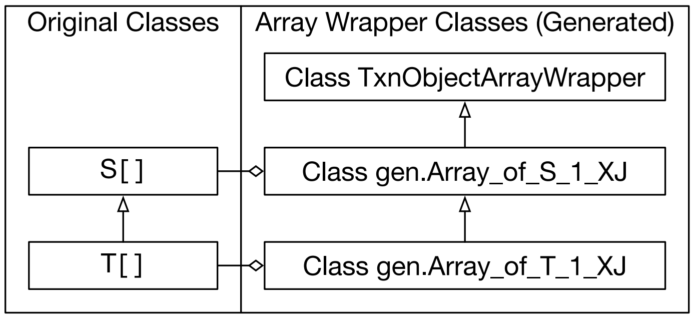

Original Classes
Array Wrapper Classes (Generated)
Class TxnObjectArrayWrapper
S[ ]
Class gen.Array_of_S_1_XJ
T[ ]
Class gen.Array_of_T_1_XJ

Figure 4.3.: Transformation of array types

transform the constructors such that any call to a superclass constructor is replaced with a call to the dummy constructor in the superclass. Also, within the corresponding `get_$init` method for that constructor we insert a call to the corresponding `get_$init` method of the superclass constructor. This transformation enables us to create an empty object first, and then run all the code of the constructor within the boundary of a transaction.

4.6.4 Java Agent

The load-time bytecode rewriter is a Java agent built using the Java Virtual Machine Tool Interface (JVMTI). It runs as a separate process, and can rewrite all loaded classes (including those loaded by the bootstrap class loader). However we do not rewrite all classes, rather we maintain a clean separation between application code that is rewritten for transactional execution and the run-time library code that *supports* transactions avoids entanglement and complexity. There is no need to produce code that must be made to serve in both the run-time and the application contexts, with the associated run-time overhead needed to distinguish the context. For bootstrap classes we generate a new version of the class under a different package name, while also preserving the original class. The rewritten class has no relationship to the original, other than that its source was the original class. Application classes (loaded by the application class loader) are rewritten to refer to the new bootstrap classes rather than the originals, while the transactional run-time library classes, being infrastructural in nature, continue to use the original versions. This creates a clean separation between the run-time library and the application.

We use the ASM library [3] for instrumenting Java classes. The instrumenter process is created in the `Agent_OnLoad` function. The agent uses the `ClassFileLoadHook` callback to intercept classes loaded by the JVM. Intercepted classes are then presented to the instrumenter. The agent communicates with the instrumenter via pipes. The result of this instrumentation process could be a single class or multiple classes. If the result is a single class, the agent returns the bytes received from the instrumenter as the bytes of the instrumented class. If the result consists multiple classes, any additional class files are injected into the VM via the `DefineClass` JNI function. In both cases the bytes of the original loaded class are replaced by the rewritten bytes. Calling the `DefineClass` function on the additional classes inside the agent causes those class definitions to be intercepted again (because of the `ClassFileLoadHook`), but there is no need to call the instrumenter for them because the agent already has their instrumented versions. To support this functionality the agent keeps a local cache for any additional class files obtained from the instrumenter. When the agent intercepts the loading of any class, it first checks if the class already has an instrumented version in the local cache, and if so, it uses that version instead of invoking the instrumenter and then removes the class from the cache. Otherwise it sends the class to the instrumenter for instrumentation as usual.

4.6.5 Instrumenter Process

The instrumenter runs in a infinite loop polling for messages by the agent. The first byte of each message from the agent is a code indicating the action requested from the instrumenter. This control byte indicates the class loader of the object (the bootstrap class loader or not), or that the VM has been initialized or is being shut down. Once a request is received for instrumenting a class, the instrumenter performs rewrites based on its classification. The preprocessed information is used to determine the classification of a class.

Although we divide classes into two categories, the general rewrites we perform on individual elements of these classes are similar. We now describe those rewrites.

- Generate a static singleton for the given class
- Generate accessor classes for each field in a class. The accessor classes are used for logging reads/writes in STM methods as explained in Section 4.6.6
- The first rewrite we do is to redirect to newly generated types. This includes redirecting to wrapped versions of objects and rewriting `getstatic`, `putstatic`, and `invokestatic` to refer to static singletons. We also redirect to the newly generated array and accessor classes
- Transform constructors as described in Section 4.6.3
- Create transactional versions of all methods as described in Section 4.7.

4.6.6 Accessor Objects

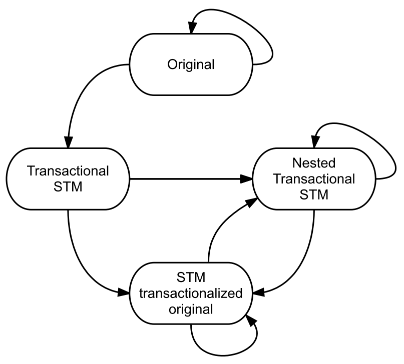
We generate accessor classes for each field of a class. Each generated class extends `org.ruggedj.xj.xjrt.runtime.Accessor`, which has a single abstract method called `restoreField` used by the run-time library to perform undo operations. It takes a `TxnWriteLog` as an argument and returns `void`. The generated accessor class also has a `set` method for setting the value of the field and a `get` method corresponding to its data type for getting the value of the field. The `set` method pushes the object being updated into the write log along with the accessor instance and the value been written. The corresponding `get` method pushes the current object into the read log along with the value being read. [2] The generated accessor class instances are created in the static initializer of a class and held in new static final fields. During the instrumentation phase, `getfield`, `putfield`, `getstatic`, and `putstatic` bytecodes are rewritten to use the accessor object for logging the field prior to setting and getting a field. The `restoreField` method pops the object

[2]Our current system records only a version number of the whole instance, but the API allows for a wide range of transaction management strategies.

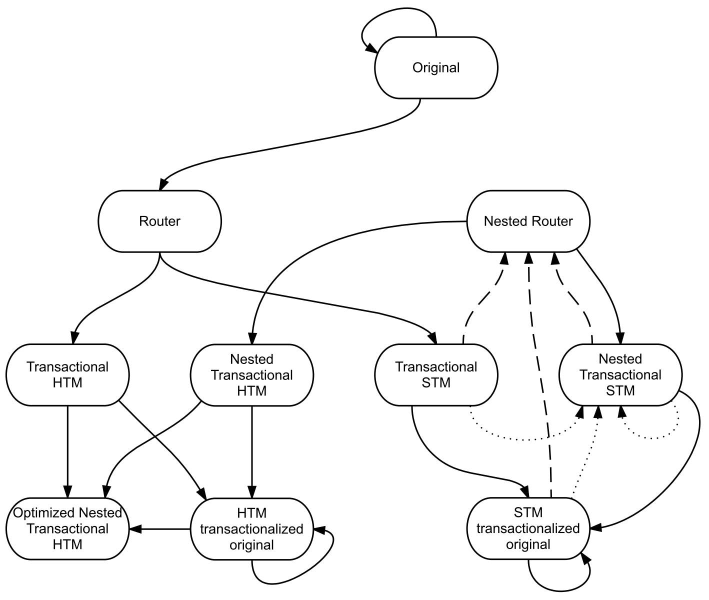
from the write log and then pops a value of the corresponding data type from it (one of the primitive types or `Object`). It then sets the field of the class to the popped value (cast to the field's declared type), restoring its value. The run-time library provides accessor classes for array types, one for each primitive array type, and one for object arrays, so these do not need to be generated for each type.

4.7 Transactional Methods in XJ

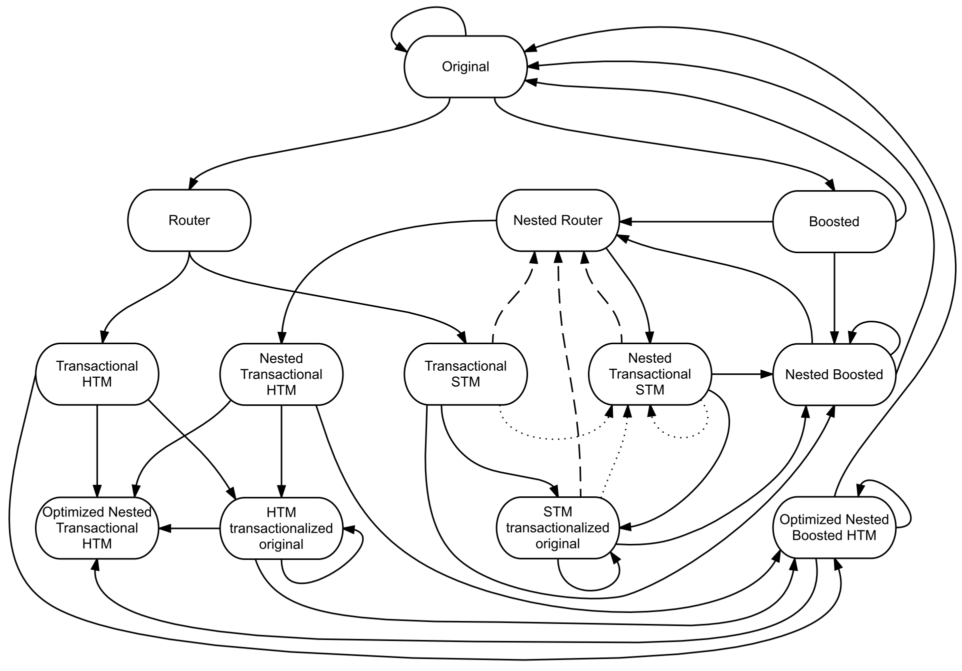
The XJ framework has evolved through the orderly addition of various transaction constructs to the base Java platform. When we first conceived XJ we planned on implementing at least STM in the form of atomic blocks and methods as first proposed by Harris and Fraser [23], using rewriting techniques similar to those of Hindman and Grossman [28]. We also planned to support both closed and open flavors of *linear* nesting [40; 41; 42], which allows a *parent* transaction to invoke a sequence of sub-operations, some of which may themselves also execute as *child* subtransactions, and where only one child is ever active at a time. As such, we envisioned the need for method variants that included: (1) the original non-transactional methods, unmodified, for execution outside transactions, (2) transactional code for atomic methods invoked as top-level transactions, containing the machinery to start a new transaction and control its outcomes (abort or commit), (3) transactionalized variants for non-atomic methods, modified for invocation from transactional contexts, and (4) transactional code for atomic methods invoked as nested transactions, containing the machinery to start and control a new nested transaction. Along with these variants came rules as to which variant can invoke which. For example, non-transactional methods can initiate a top-level transaction by calling an atomic method. These in turn may invoke non-atomic methods, but the variant invoked must contain machinery for transactional execution. And nested invocation of an atomic method from an existing transaction context must dispatch to a nested variant of that atomic method. We capture these rules in the form of the state transition diagram shown in Fig. 4.4a.



(a) STM method variants



(b) STM and HTM method variants



(c) The full set of XJ method variants

Figure 4.4.: Method call diagrams. Solid arrows indicate transitions based on the transaction semantics of the callee. Dotted arrows indicate transitions from closed transaction contexts. Dashed arrows indicate transitions from open transaction contexts.

Having implemented this scheme for STM, we also desired for comparison between open nesting and the related *boosting* approach of Herlihy and Koskinen [24]. This resulted in two additional method variants, Boosted and Nested Boosted. The first was to be called from a non-transactional context while the other was to be called from a transactional context. Their addition to the system was an orthogonal change and did not significantly affect anything in the STM variants other than the addition of boosted forms to parallel the open nested forms.

Experience with STM and the arrival of commodity best-effort HTM support with Intel's Haswell processors led to our devising a scheme for HTM-accelerated transactions. This addition was more complicated than adding boosting. We had to consider how STM could execute alongside HTM and also what it would mean for HTM to be called from STM and vice versa. The fact that HTM was *best effort* threw more complexity into the mix. This meant that we needed to devise mechanisms for backing off from HTM to STM in the face of failures, while prioritizing use of HTM whenever possible (to maximize performance gains). Fig. 4.4b summarizes the changes we made to include HTM in the system (for simplicity we do not include boosting here). The main additions to the state diagram, in addition to HTM variants of the methods, are the Router and the Nested Router. These generated variants are responsible for routing a method call to either its HTM or STM version, based on run-time heuristics. With the explosion of states occurring when boosting is included, we found the resulting state diagram in Fig. 4.4c essential to our understanding of the rules for generating code for these method variants. The following sections describe this resulting methods of the XJ framework by reference to Fig. 4.4c. Chapter 6 then considers how this formulation allows us to consider additional extensions to XJ.

4.7.1 Generated Method Variants

The XJ system can generate 13 different method variants. The variants generated depend on the transactional semantics exhibited by the original source code. These semantics can be divided into four main categories:

Table 4.1: Generated method variants by source type

Source method type	**Generated method types**
Non transactional (NT)	Original (O) STM transactionalized original (STO) HTM transactionalized original (HTO)
Closed atomic (CA) Open atomic (OA)	Router (R) Nested router (NR) Transactional STM (TS) Nested transactional STM (NTS) Transactional HTM (TH) Nested transactional HTM (NTH) Optimized nested transactional HTM (ONTH)
Boosted (B)	Boosted (B) Nested boosted (NB) Optimized nested boosted HTM (ONBH)

- **Non-transactional (NT)** methods do not have any specific transaction semantics associated with them. They do not create new transactions. A majority of methods fall into this category.
- **Closed atomic (CA)** methods exhibit *closed* transaction semantics. They create new closed atomic transactions. In XJ, these are methods that either belong to a class marked `xatomic` or have the `xatomic` method modifier, or methods that contain an `xatomic` block.
- **Open atomic (OA)** methods exhibit *open* transaction semantics. They create new open atomic transactions. Methods that belong to classes marked `openatomic` are treated as open atomic methods.
- **Boosted (B)** methods exhibit *boosted* transaction semantics. They create new boosted transactions. In XJ, these are methods that belong to a class marked with the `boostedatomic` class modifier.

Not all variants are generated for a given source method—only the relevant method variants are generated. Table 4.1 shows the method types generated for each source method type.

These generated method variants create a complex system with method variants appropriately calling each other. The transaction semantics of the source method type induces some restrictions on what method variants it can call. Figure 4.4c captures these restrictions and presents them in a structured manner in the form of a state transition diagram. In this diagram, we represent each generated method variant as a state, with arrows showing variants they are permitted to call (arrows go from caller to callee). Most states have multiple outgoing arrows; the particular one taken depends on the transaction semantics of the method being called. Most outgoing arrows in this diagram are solid arrows. This indicates that the transition is solely based on the transaction semantics of the method being called. The dotted and dashed arrows indicate that the transition taken depends also on the transaction context of the parent transaction. The transition indicated by the dashed line is taken if the parent of the current transaction is an open transaction, while the dotted line is taken if the parent is a closed transaction. The reason for this is subtle, and is explained in Section 4.2. The rules for all transitions, along with information on what makes these generated methods unique, appears in Table 4.2.

XJ keeps a copy of the original method, for use in non-transactional contexts. Such original methods are not subject to any TM-specific instrumentation. However, they do undergo the general XJ rewrites described in Section 4.6.

Except for the Router and Nested Router variants, all generated methods derive from the original method. They can be thought of as transactional versions of the original method, with instrumentation added to carry out various transaction semantics. We refer to these versions of the generated methods as *transactional versions* of the original method. Transactional versions include the STM, HTM, and boosted versions of the original method. All STM versions of the methods contain STM-specific instrumentation, as described in Section 4.7.2. Section 4.7.3 describes the HTM-specific rewrites applied to all HTM versions of the method. Boosted versions of the method do not have much instrumentation added. STM, HTM, and boosted methods called from a non-transactional context create a new TxnDescriptor, while nested transactional methods use the TxnDescriptor passed to them by their parent. Various mechanisms can be used to achieve this

Table 4.2: Transitions and actions of method variations

Method type	**Method call (callee)**				**Object locking**		**Field logging**	**Abstract locks**	**Txn ctrl**	**New TD**
	NT	**CA**	**OA**	**B**	**get**	**set**				
Original (O)	O	R	R	B	None	None	None	–	No	–
Router (R)	–	TH/TS	TH/TS	–	–	–	–	–	No	–
Nested router (NR)	–	NTH/NTS[a]	NTH/NTS[a]	–	–	–	–	–	No	–
STM transactionalized original (STO)	STO	NTS/NR[b]	NTS/NR[b]	NB	Record	Lock	Log	–	No	–
Transactional STM (TS)	STO	NTS/NR[b]	NTS/NR[b]	NB	Record	Lock	Log	Obtain	Yes	Yes
Nested transactional STM (NTS)	STO	NTS/NR[b]	NTS/NR[b]	NB	Record	Lock	Log	Obtain	Yes	No
HTM transactionalized original (HTO)	HTO	ONTH	ONTH	ONBH	Check	Inc version	None	–	No	–
Transactional HTM (TH)	HTO	ONTH	ONTH	ONBH	Check	Inc version	None	Check	Yes	Yes
Nested transactional HTM (NTH)	HTO	ONTH	ONTH	ONBH	Check	Inc version	None	Check	Yes	No
Optimized nested transactional HTM (ONTH)	HTO	ONTH	ONTH	ONBH	Check	Inc version	None	Check	No	No
Boosted (B)	O	NR	NR	NB	None	None	None	Obtain	Yes	Yes
Nested boosted (NB)	O	NR	NR	NB	None	None	None	Obtain	Yes	No
Optimized nested boosted HTM (ONBH)	O	ONTH	ONTH	ONBH	None	None	None	Check	No	No

[a] NTH/NTS: Call HTM or STM depending on backoff policy.
[b] NTS/NR: Call NTS if parent is closed, NR is parent is open.
– indicates not applicable.

goal; in XJ we pass the TxnDescriptor as the last argument to the method. Except for the optimized version of the nested HTM method, all other transactional methods contain instrumentation to perform transaction control as described in Section 4.7.4.

In contrast to the other generated methods in the system, which are derived from the original version of the method, the Router and the Nested Router are synthetic methods. As their name suggests, these methods are responsible for routing the intended method call to either the HTM or STM version of the method. This decision is driven by a self-tuning back-off policy that attempts to use HTM whenever it is likely to commit (so as to improve performance) and to avoid HTM when it is unlikely to succeed.

4.7.2 STM Specific Rewrites

Table 4.2 shows that STM related methods need several constructs instrumented to carry out transaction semantics. As previously described, getting a field requires that the transaction check whether the containing object is locked by some other transaction. If it is locked by another transaction, the current transaction aborts. (In our system, it throws an exception that is caught by a Java exception handler placed around the transaction, which then uses the transaction's log to undo the effects of the transaction.) If the containing object is not locked at all, then the version number of the object is noted in the log for later validation at transaction commit time. Finally, if the object is locked by the current transaction, no further locking work is needed. We implement these locking actions in a run-time method called `openForRead`. After the locking actions the field is read as usual.

When a transaction desires to set a field it once more checks the version/lock word. If the current transaction has locked the containing object, no further locking work is needed. If a different transaction has locked the object, then the current transaction aborts. Finally, if the object is not locked, the current transaction attempts to acquire the object's lock with an atomic compare-and-swap operation. If it succeeds, it records that fact in its log. If it fails, the transaction aborts. This locking work is performed by the `openForWrite` run-time method. Before setting the field, the transaction records the current value of the field in its log. Then it sets the field.

In the case of open nesting, the STM transaction will also attempt to acquire the specified abstract locks, aborting if it cannot, and recording the locks and any abort, etc., handlers in the log.

4.7.3 HTM Specific Rewrites

HTM versions of the method rely mostly on hardware to take care of transactional semantics. However, we must guarantee that HTM transactions will inter-operate properly with STM transactions, so we have HTM versions of `openForRead` and `openForWrite`. The `openForReadHTM` run-time method reads the version/lock word, and explicitly aborts the current transaction if the object is locked by some other transaction. It does not need to record the version number in the log. The `openForWriteHTM` method also examines the version/lock word. If the object is locked by the current transaction, then it does nothing; if it locked by some other transaction, it explicitly aborts; and if it is not locked, it increments the version number. Again, there is no need to log the current value of the field being set.

In the open nesting and boosted cases, HTM does not need to acquire abstract locks. However, it does need to verify that it *could* have acquired them. This is better than actually acquiring and releasing them. Not only is it less work, but it involves only reading the abstract lock data structure, not updating it, avoiding needless conflicts on that data structure.

```
method_stm (args /*, txnDescriptor if nested */) {
 TxnDescriptor txnDescriptor = TxnDescriptor.newTxnDescriptor() // If not
    nested
 numRetries = 0;
 succeed = true;
 while (true) {
   try {
     succeed = true;
     beginTxn(txnDescriptor);
     try {
       obtain abstract_locks // if open/boosted
       // STM instrumented method body
       ...
     } catch (txnException) {
       succeed = false;
       abortTxn(txnDescriptor, ++numRetries);
       continue;
     } catch (Error) {
       succeed = false;
       abortTxn(txnDescriptor);
       throw Error;
     }
   } finally {
     if (succeed) {
       try {
         Log handlers // if open/boosted
         commitTxn(txnDescriptor);
       } catch (TxnException) {
         abortTxn(txnDescriptor, ++numRetries);
         continue;
```

```
        }
      }
    }
  }
}
```

Listing 4.4: Pseudo-code for the STM version of a method

4.7.4 Transaction Control

Most transactional methods in XJ require transaction control code. Transaction control includes loops for handling retries of transactions, aborting transactions in the face of conflicts, the back-off policy, and logging handlers when executing open and boosted transactions. Listing 4.4 shows the transaction control for an STM version of the method. The back-off policy is not embedded in the generated code; the run-time method that implements `abortTxn` takes care of it. Note that while the listing shows Java pseudo-code, XJ actually inserts the bytecode equivalent when rewriting.

```
// i and j are unique for each transaction
static i = 1;
static j = 0;
method (args /*, txnDescriptor if nested */) {
  if (j == 0) {
    method_htm(args /*, txnDescriptor if nested */); }
  else {
    j = j - 1;
    method_stm(args /*, txnDescriptor if nested */);
  }
}
```

Listing 4.5: Pseudo-code for the routing method

```
method_htm (args /*, txnDescriptor if nested */, boolean runInSW) {
  TxnDescriptor txnDescriptor = TxnDescriptor.getNewDescriptor() // if not
      nested
  numRetries = 0;
  txnStatus = 0;
  while (true) {
    try {
      txnStatus = TxnDescriptor.beginOpenHtm(txnDescriptor, runInSW);
      if (txnStatus == -1) { // running in HTM mode
        try {
          check abstract_locks // if open/boosted
          // HTM instrumented method body
          ...
        } catch (TxnException) {
          TxnDescriptor.abortOpenHtm(txnDescriptor, numRetries, runInSW);
        } catch (Error) {
          TxnDescriptor.abortOpenHtm(txnDescriptor, runInSW);
        }
      } else if (WARMUP_PHASE) {
        if (TxnDescriptor.retryInSWMode(txnStatus, numRetries) {
          method_htm(args, true);
          return;
        }
        numRetries++;
        continue;
      } else { // HTM failed
        if (TxnDescriptor.retryInHW(txnStatus, numRetries) {
          ++numRetries;
          continue;
        }
```

```
          // Back off to SW mode
          j = i;
          i = i * 2;
          method_stm(args);
          return;
        }
      } finally {
        if (txnStatus == -1) {
          try {
            Log handlers // if open/boosted
            TxnDescriptor.commitOpenHtm(txnDescriptor, runInSW);
            j = 0;
            return;
          } catch (TxnException) {
            TxnDescriptor.abortOpenHtm(txnDescriptor, numRetries, runInSW);
          }
        }
      }
    }
  }
```

Listing 4.6: Pseudo-code for the HTM version of a method

Transaction control for HTM methods is a little more complicated: it also needs to back off to the STM version of the method, something that the STM version of the method need not do. This is shown in Listing 4.6. Although it looks similar to the STM version of the method, it has subtle differences. First, the run-time version of the method to start a hardware transaction returns a value. The returned value is -1 if the transaction was created successfully (an aborted transaction will never return a value of -1). When an HTM aborts, this method will return the status code of the aborting transaction. Bits of this value can be inspected to see the underlying reason for failure. This is used by the `retryHTM` method to decide if the transaction should to be retried in HTM or STM. Secondly, the `i` and `j` values in this version of the method are used to auto-tune the back-off from HTM to STM. These are static variables that are unique to each transaction. Also, `i` and `j` are packed into a single word, allowing both to be update with one store (not shown here). The back-off policy we implement is a self-tuning scheme where we give priority to running under HTM (because it gives better throughput) but falls back to STM when HTM fails (the number of times to retry in HTM is driven by the result code of the aborting HTM transaction). After an HTM method fails, it updates the `i` and `j` values that enable the router to take a better decision the next time around. The router consults the `i` and `j` values when directing the method call to either the HTM or STM version of the method, as shown in Listing 4.5.

As Table 4.2 shows, there are a few transactional methods that do not have transaction control added to them. The transactionalized original versions of the method do not have any transaction control because they do not create any transactions. They are merely the original version of the method with instrumentation for locking and logging where applicable. It should also be noted that although Optimized nested transactional HTM and Optimized nested boosted HTM create nested transactions, they do not have any transaction control. This is because they are optimized versions of nested HTM transactions that are always called from a transactional context running under HTM. As mentioned before our HTM implementation relies on Intel TSX instructions, and thus nested transactions under HTM will always be flattened. We recognize this and generate these optimized versions of the method with no transaction control.

For each of the generated transactionalized versions of the classes we apply a series of transformations. These transformations generate new methods, as well as transform existing methods in order to add the transactional machinery. In general, transactional programs can run with or without HTM support. When the system is run with support for HTM, the bytecode instrumenter performs a series of additional method transformations. We generate a transactional version of the method for both HTM and STM. The STM version calls runtime routines that support STM, while the HTM versions call routines that support HTM. Examples of such run-time calls are `openForRead` (indicating that an object is about to be read), `openForWrite` (indicating that an object is about to be written), `beginTxn`, `endTxn`, etc. We also generate a *routing method* that direct the caller to either the STM or HTM version of the method, with the decision guided by two special variables we generate for each routing method. Listing 4.5 shows pseudo-code for the routing method, and Listing 4.6 for the HTM version of a method. Although these show `i` and `j` as static members of the class, in reality they are encapsulated in a separate object referenced from a `static final` field. This guarantees that updating of `i` and`j` will not cause false conflicts with other transactions. The idea behind the scheme is that we first try to run a transaction in HTM. In the face of HTM failures we back off to STM aggressively, yet try HTM once in a while. If a transaction succeeds in HTM, then we will keep trying it in HTM.

This simple scheme worked well when the number of threads was low, but failed to yield its true potential as the number of threads increased. It was backing off too aggressively and not attempting enough times in HTM, which limited the throughput that we could achieve. To remedy this, we introduced thread-local counters so that most counting down occurs per-thread rather than against shared counters. We call the thread-local counters `decrementCounter` and `updateCounter`, and they are initialized to 10. Each time the shared `j` would have been decremented, we check if the thread-local `decrementCounter` is at 0, and if so we decrement the shared `j` and reset `decrementCounter` to 10. Otherwise we just decrement the thread-local `decrementCounter`. The same goes for updating `i` and `j` when backing off from STM to HTM, using `updateCounter` in place of `decrementCounter`. This scheme ensures that the back-off rate does not change drastically as the number of threads increases. The scheme we use is much simpler than that of Diegues and Romano [18], who used a reinforcement learning technique to decide when to use the fallback path for TSX.

One of the main issues we encountered early on with using HTM was that many transactions failed with result code 0 (i.e., no specific reason given). Using the Intel SDE, we found these aborts to be caused by execution of instructions that are incompatible with TSX [29]—`FXRSTOR` and `FXSAVE` (perhaps among others)—and which are compiled into HotSpot's run-time stubs used to control dynamic optimization and linking, and to resolve

Java static and virtual method calls. By design, HotSpot patches these call sites at run time [43]. Thus our hardware transactions always failed, *and* those failures preventing triggering of the patching mechanism. Our workaround was to devise a mechanism to "warm" the system up in STM mode before attempting any hardware transactions. However, so that the compiler's optimizations will be triggered appropriately, and so that linking/patching will occur, these STM transactions must follow the same code path (except for not using the `XBEGIN` instruction, etc.) as HTM transactions do. We use a global flag to indicate whether we are in the software-only warm up phase. The `i` and `j` values described above are used only after warming up.

The bytecode rewriter generates code that preserves many important invariants related to possible transitions between STM and HTM code. We follow a few simple rules. The HTM version of a method always calls the nested HTM version of other methods. The nested HTM version of a method is much simpler than the one presented in Listing 4.6. Since nested HTM methods will always be called from an HTM context, they do not need to begin a new transaction, and thus they contain only the instrumented method body. On the other hand, the STM version of the method calls the method with the original name. Thus, if the method being called is a transactional method, it will call the routing method. This enables the new transaction to run under *either* HTM or STM. There is one caveat though: a parent transaction running under STM should not create a nested closed hardware transaction (since it will not gather locks and log records and accrue them to the parent).[3] In contrast, if the parent transaction is running under STM then an *open* nested child transaction can safely run under HTM. This is because the open nested transaction will acquire abstract locks and undos and can release all physical locks (making HTM possibly profitable in this case). We acquire the abstract locks and log undos before starting the hardware transaction, and release/revert them if the hardware transaction fails. We further optimize the case where an *open* nested action runs in hardware under a top-level hardware transaction. Such a child does not need actually to acquire locks or log undos, since they will be immediately discarded on either success or failure of the hardware trans-

[3]Doing so is possible, but would mean the HTM version does all the work of the STM version, with the added overhead of starting and committing an HTM transaction.

action. However, to detect conflicts, the child must check that it *could* have acquired the locks—i.e., that there are no conflicting locks held by other transactions.

4.8 Run-Time Library

The run-time library provides the dynamic support needed for transactional execution. It supports both closed and open nested transactions, running under HTM or STM simultaneously, as well as boosting. Thus a program can make use of all styles of transactional execution. Our experiments also configure the run-time library for modes of execution that support only one of closed, or boosted transactions, so as to isolate the overheads for each mode. For example, the data structures needed for tracking the reads and writes of open/-closed nested transactions are not needed for boosting and there is no need to instantiate them in that case.

The run-time library offers both HTM and STM versions of all important methods. As previously explained, the HTM version of the method takes an additional `boolean` argument indicating whether it should run in "software mode." The run-time library also maintains statistics in a thread-local manner, avoiding false conflicts in the statistics collection process.

The library performs conflict detection at the level of objects, and tracks writes at the level of fields using an undo log. Each transactionalized object carries an extra field, which holds the lock for writes, and otherwise contains a version number for the object, which is incremented upon commit. In our implementation HTM and STM can safely co-exist simultaneously. Thus the two mechanisms need to play well with each other. In general, we adopt pessimistic concurrency control for writes, and optimistic concurrency control for reads. When running under STM, writes acquire a lock on the object. Reads proceed optimistically, simply logging the value of this field (a version number), and the log is then processed at commit time to validate the transaction (if the logged version number does not match the current value and the owner of a locked object is not the current transaction (or an ancestor) then the transaction aborts). When running under HTM, writes simply increment the version number, thus invalidating conflicting STM readers and conflicting with HTM readers or writers. Reads under HTM perform a check to make sure that the object is not locked by a non-ancestor transaction, explicitly aborting if necessary. In sum, the lock/version word "glues" together the STM and HTM schemes into a coherent (and safe!) hybrid TM.

The implementation of `PointSpace` that we use in our experiments itself requires a concurrent data structure to store the lock metadata because multiple transactions can try to acquire abstract locks concurrently. We use the `NonBlockingFriendlyHashMap` of Crain et al. [11] for this purpose.

5 EVALUATION

5.1 Workload

Our workloads extend Synchrobench [21], which is a micro-benchmark suite for evaluating synchronization techniques on collection classes such as sets and maps. It provides implementations for a variety of differently synchronized data structures in Java (as well as C/C++). It defines abstract APIs comprising simple `add`, `remove`, `contains`, and `get` operations that the data structures must implement. Adding new implementations to the framework is simply a matter of making them conform to one of these APIs. The `CompositionalIntSet` interface abstracts sets, while `CompositionalMap` abstracts maps.

We extend Synchrobench for use with nested transactions in several ways. First, we provide open atomic, closed atomic, and boosted implementations of the `CompositionalMap` and `CompositionalIntSet` interfaces in our language dialect. These classes are compiled by our modified compiler. We also augment the Synchrobench driver to instantiate these implementations for measurement. Second, we reconfigure the driver to run *transactions* of various sizes, consisting of aggregate operations on the underlying data structures. This enables benchmarking for throughput while varying transaction granularity. Third, we reconfigure the driver to offer the ability to *pin* worker threads to specific cores. Finally, we make refinements to the manner in which the driver calculates throughput numbers. We now describe these modifications in more detail.

5.1.1 Open Atomic Workload

Listing 5.1 shows the `OpenIntSet` class, which is an open atomic implementation of `CompositionalIntSet`. `OpenIntSet` provides a concurrency-safe wrapper for unsynchronized implementations of `CompositionalIntSet`. Similarly, `OpenMap` provides a concurrency-safe wrapper for unsynchronized `CompositionalMap` implementations. Here we give more precise details of the implementation of `OpenIntSet`; `OpenMap` is derived similarly.

As in the earlier `OrderedSet` example, `OpenIntSet` defines two distinct lock spaces: `eltSpace` manages abstract locks issued on points corresponding to elements in the set, and `setSpace` defines abstract locks for the set as a whole. The `addInt` method attempts to add the element `elt` to the set. Thus it needs an `X` lock on the point represented by element `elt` from the `eltSpace` lock space, and a `C` lock for the set as a whole from the `setSpace` lock space.

Generally, `onabort` handlers are needed only for methods that change the abstract state of the set. One such method is `addInt`, which returns `true` if the element was added to the set and `false` if the element was already present. Thus its `onabort` handler must remove the element from the set only if it was not previously there. To achieve this, the `onabort` clause captures and uses the `result` of the committed body of the method. The other methods can be derived similarly. Our extended transactional Java syntax supports declarations for variables (like `result`) outside the body of the open atomic method that are visible to the body and the `onabort` clause.

5.1.2 Closed Atomic Workload

The `ClosedIntSet` class shown in Listing 5.2 provides a concurrency-safe wrapper, using closed nesting, for an unsynchronized `CompositionalIntSet`. The methods of `ClosedIntSet` execute the set operations in (closed) nested mode.

5.1.3 Boosted Workload

Boosted and open atomic classes look similar since they both must make use of abstract locks to protect the abstract state of the underlying data structure. Listing 5.3 shows `BoostedMap` as an implementation of the `CompositionalMap` interface. Unlike an open

```
public openatomic class OpenIntSet
    implements CompositionalIntSet {
  private final CompositionalIntSet intSet;
  private final
    LockSpace
      <SXMode,PointSpace<SXMode,Integer>>
    eltSpace
      = new PointSpace<SXMode,Integer>();
  private final
    LockSpace
      <PCMode,UnitSpace<PCMode,OpenIntSet>>
    setSpace
      = new UnitSpace<PCMode,OpenIntSet>();
  public
    OpenIntSet(CompositionalIntSet intSet)
    { this.intSet = intSet; }
  public boolean addInt (int elt)
    [boolean result = false]
    locking
      (eltSpace : point(elt) : SXMode.X),
      (setSpace : get() : PCMode.C)
    { return (result = intSet.addInt(elt)); }
    onabort
    { if (result) intSet.removeInt(elt); }
  // etc.
}
```

Listing 5.1: `OpenIntSet` class

```
public xatomic class ClosedIntSet
    implements CompositionalIntSet {
  private final CompositionalIntSet intSet;
  public ClosedIntSet(CompositionalIntSet intSet)
    { this.intSet = intSet; }
  public xatomic boolean addInt(int x)
    { return intSet.addInt(x); }
  // etc.
}
```

Listing 5.2: `ClosedIntSet` class

```
public boostedatomic class BoostedMap<K,V>
    implements CompositionalMap<K,V> {
  private final ConcurrentMap<K,V> map;
  private final
    LockSpace<SXMode,PointSpace<SXMode,K>>
    keySpace = new PointSpace<SXMode, K>();
  private final
    LockSpace
      <PCMode,UnitSpace<PCMode, BoostedMap<K,V>>>
    mapSpace
      = new UnitSpace<PCMode, BoostedMap<K,V>>();
  public BoostedMap(ConcurrentMap<K,V> map)
    { this.map = map; }
  public V put(K key, V val)
    [V result]
    locking
      (keySpace : point(key) : SXMode.X),
      (mapSpace : get() : PCMode.C)
    { return (result = map.put(key, val)); }
    onabort {
      if (result == null) map.remove(key);
      else map.put(key, result);
    }
  // etc.
}
```

Listing 5.3: `BoostedMap` class

atomic class which wraps an unsynchronized implementation, a boosted class wraps a thread-safe implementation of the `CompositionalMap` interface. This is an important distinction.

5.1.4 Support for Varying Transaction Sizes

We extend the driver for Synchrobench to aggregate some number of underlying data structure operations nested within a top-level closed transaction, parameterized by a new run-time flag g. We modified the worker threads of Synchrobench accordingly as shown in Listing 5.4. If the parameter g has a value greater than 0 then the operations are performed

```
private xatomic void atomicDoOperation() {
  for (int i = 0;
       i < Parameters.groupSize;
       i++)
    doOperation();
}
```

Listing 5.4: Top-level transaction for nesting

```
@Atomic(metainf = "elastic")
private void deuceAtomicDoOperation() {
  for (int i = 0;
       i < Parameters.groupSize;
       i++)
    doOperation();
}
```

Listing 5.5: Top-level transaction for Deuce

within a top-level closed transaction by marking `atomicDoOperation` as `xatomic`. Then `doOperation` will be nested/boosted accordingly within the top-level transaction. We also compare against Deuce [33], for which we use the corresponding method shown in Listing 5.5, to achieve the same effect.

5.1.5 Support for Thread Pinning

We update the driver for Synchrobench to accommodate the option of specifying a strategy for pinning worker threads. The new run-time flag `ps` can be used to specify this strategy. The value accepted is any combination of the characters `C`, `S`, and `H`. The character `C` represents core, `S` represents socket, while `H` represents hyperthread. These characters represent the 3 different dimensions that can be varied when pinning threads. The sequence of the characters specifies which aspect of these to vary most rapidly when pinning threads. For example, `CSH` means to vary the core first, then the socket, and finally hyperthreads of the same core.

```
@Atomic // API method
public boolean addInt(int x) ...
// Methods used by the maintenance thread
@Atomic(metainf = "maint")
private Node getChild(Node n, boolean left) ...
```

Listing 5.6: Deuce STM implementation of `TFTreeSet`

```
// API method
public boolean addInt(int x) ...
// Methods used by the maintenance thread
private xatomic
Node getChild(Node n, boolean left) ...
```

Listing 5.7: Transactional implementation of `TFTreeSet`

5.1.6 Modified Transaction Friendly Data Structure

Synchrobench [21] contains transaction-friendly data structures that are "speculation-friendly" [12]. We took the transaction-friendly `TreeSet` binary search tree implementation and modified it to run with transactions. We refer to this as `TFTreeSet`. It uses a separate maintenance thread to keep the data structure properly balanced. Inserts are done at the leaf level, while deleting an element simply marks the node as deleted. The maintenance thread rebalances the data structure and removes deleted nodes. In the implementation for Deuce the maintenance thread performs its tasks inside small atomic methods as shown in Listing 5.6. The API methods are also marked as atomic methods.

Adapting these data structures for our transactional Java dialect is trivial. We mark those methods used by the maintenance thread as closed atomic using the `xatomic` method modifier as shown in Listing 5.7. This is reasonable because the maintenance methods are short, making only a quick modification. We do not include anything special on the API methods, but leave it to our open/closed wrapper classes to enforce atomicity. Hence, depending on the wrapper that is instantiated, the API methods may run closed or open.

We also performed some hand optimizations to the benchmark code that are important in the transactional setting. These optimizations could be performed by a bytecode rewriting optimizer, a task we leave to future work. Specifically, we found places where a field is often unconditionally updated with the value it already contains. Such writes are cheap in the non-transactional case, but introduce needless conflicts in transactions. We made them conditional. We also specially mark `openForRead` and `openForWrite` calls that are redundant and `openForRead` calls that are always followed by an `openForWrite` on the same object. This substantially reduces the transactional instrumentation in the micro-benchmarks.

5.1.7 Modified Throughput Reporting for Accuracy

Previously, the Synchrobench driver thread worked as follows. It created all the worker threads, then recorded the system time, and finally started the worker threads individually. The main thread then slept for the duration of the benchmark. Upon being woken up, the main thread attempted to join all the worker threads and to record the system time again. The difference between the recorded system times is taken as the elapsed time for the benchmark iteration. Meanwhile each thread kept a record of the number of operations it executed. When reporting the results, Synchrobench divided the total number of operations completed by all the threads and divided by the elapsed time to calculate the throughput in units of operations per second. This mechanism works relatively well when running with a small number of threads, but when running on a multi-socket machine some flaws appeared. We noticed that the elapsed time when running with 48 threads was in the range of 5.5 seconds when the specified duration was 5 seconds. This had to do with the difference in the times at which each thread started (they are started one by one), and even more in the times when they stopped (after each operation, they look to see if their "stop" flag has been set; operation times vary as do the times when the "stop" flags are actually set). Thus some threads are actually idle for significant periods of time leading to an underestimate of throughput. Our remedy is to record the start and stop time of the individual worker

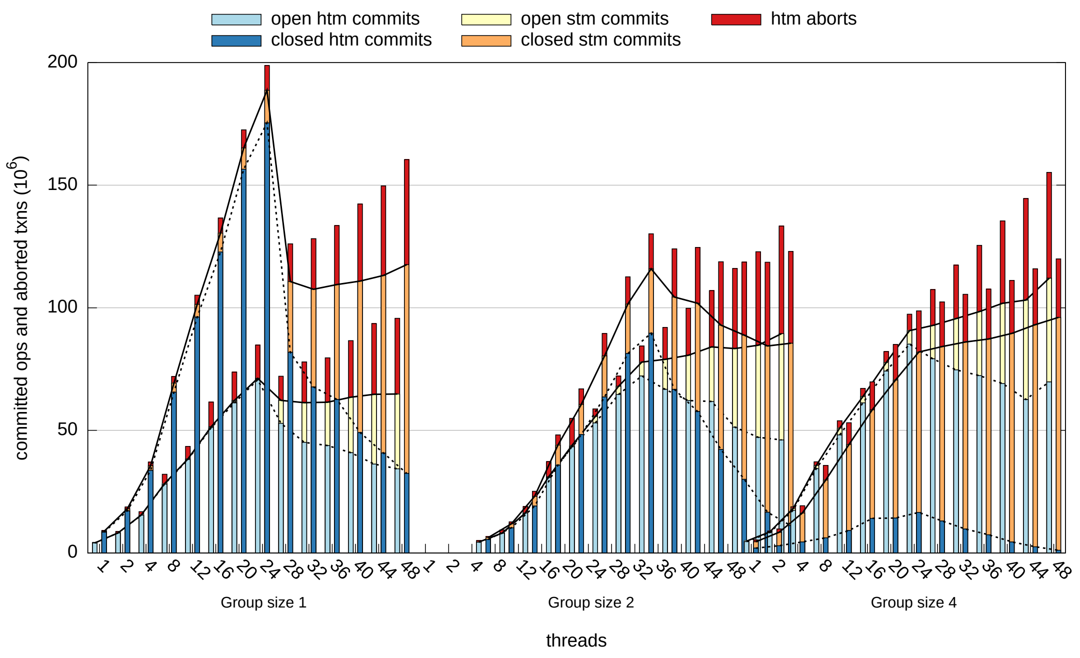


Figure 5.1.: Committed operations versus aborts

threads. We divide the total number of operations completed by the total of the running times of the worker threads, and then multiply by the number of threads. This throughput value more accurately represents average throughput for large numbers of workers.

5.2 Experiments

Our experiments explore a range of structured transactions, namely flat, closed, open, and boosted, in STM-only mode and in self-tuning hybrid HTM/STM mode. We further compare against Deuce STM, running its efficient elastic mode transactions and configured as described in Section 5.1, as a reference point. We conducted all experiments using the extended version of Synchrobench described in Section 5.1 with the parameters shown in Table 5.1.

We perform three sets of runs across these parameters so as to space the sets of five iterations over time. Thus, we sample 15 measurements for each configuration.

Table 5.1: Synchrobench parameters for experiments

$i = 16K\|64K$	The initial number of elements added to the data structure before measurement begins.
$r = 32K\|128K$	The range of possible keys used in the data structure; keys are drawn uniformly at random.
$u = 0\|5\|50$	The percent of operations that are updates, each randomly chosen either to add or remove an element.
$n = 5$	The number of iterations of the benchmark.
$t = 1\|2\|4\|8...44\|48$	The number of spawned worker threads.
$W = 5$	The warm up time in seconds that the benchmark runs before starting measurement.
$d = 5000$	The duration of a single iteration of the benchmark in milliseconds.
$g = 1\|2\|4\|8\|16\|32$	The number of operations to perform in each transaction.
$ps = CSH$	The pinning strategy to use. We first pin threads to different cores on one socket, then on the next socket, before finally assigning threads to different hyperthreads of the same core. Exploratory experiments showed this strategy to be clearly the best.

Given a benchmark data structure, Synchrobench initializes the data structure to its initial size, drawing randomly from the indicated range of values. Once the data structure is initialized, Synchrobench performs operations at random, using the update percentage to decide if the operation should be "add/remove" or "get/contains". The collected statistics are cleared once the warm up period ends, and the benchmark runs for the specified duration after that. Then Synchrobench reports statistics for the benchmark run, including the throughput (operations/s).

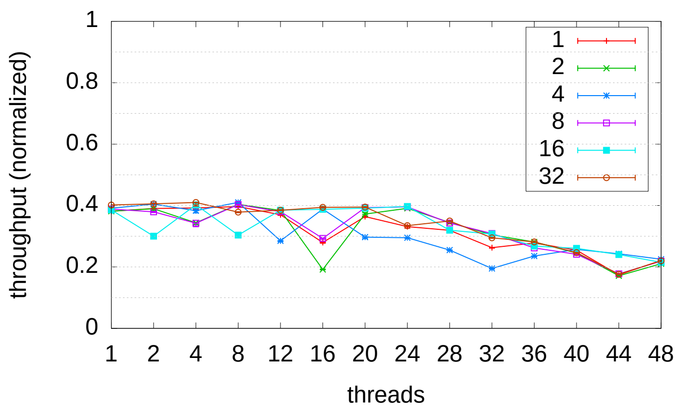
When enabling HTM, we followed a more complex warm-up procedure. First, we ran for five seconds calling the HTM routing methods of transactions. Then we paused five seconds to allow the HotSpot compiler to compile (and possibly optimize) methods. We repeated this procedure to force proper linking of the resulting compiled methods. We then forced garbage collection (so that collections will not interfere with our timings) and started Synchrobench's warm-up run. We believe that in the future this warming up approach can be generalized for arbitrary transactional applications. Alternatively, deeper modifications could be made to the compilers to make them HTM aware.

The `OpenIntSet` and `ClosedIntSet` classes are initialized with the transactionalized version of `TFTreeSet`. For boosting, `BoostedMap` is initialized with the thread-safe `NonBlockingTorontoBSTMap` [20]. For benchmarks involving Deuce STM we run `TFTreeSet` under Deuce STM.

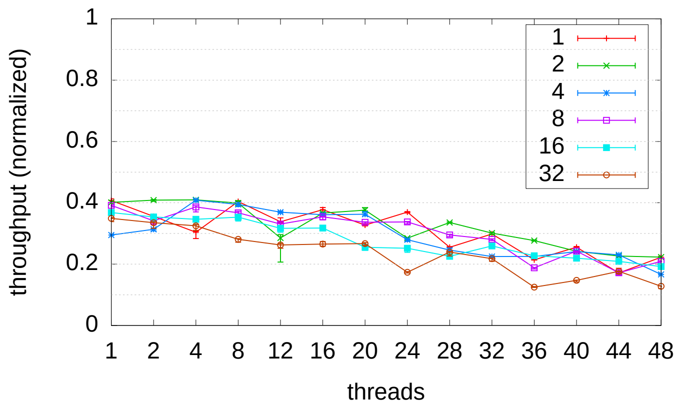
All benchmarks were run on a 48-way, x86-64 Intel Xeon E5-2690 v3 machine with 2 sockets of 12 hyperthreaded cores, with the clock frequency fixed to 2.4 GHz, and with TSX enabled. The machine was running CentOS Linux release 7.2.1511 and our modified version of OpenJDK.

5.3 Results

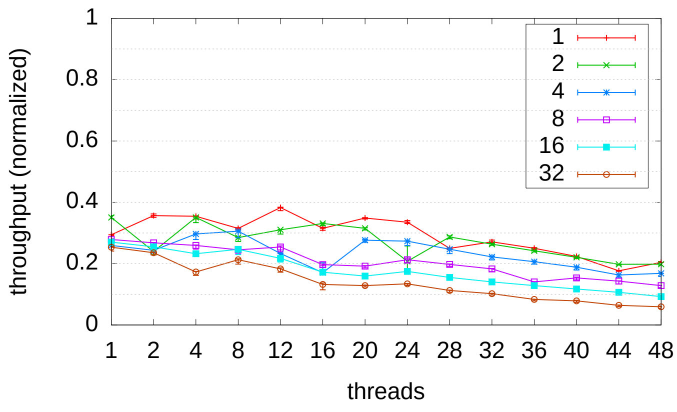
We now present results for executing the workload under different transaction implementations. Our first set of results are for data structures initialized with 64K elements and a key range of 128K. All numbers reported in throughput graphs are normalized *per-thread* throughput. This implies that perfect scaling will appear as a horizontal line in the graphs. Our normalization is relative to the standard unsynchronized "java.util.TreeMap" (run with one thread, no synchronization). At each point we plot the median along with bars showing the 10th and 90th percentiles across the 15 total iterations we accumulated. A common theme in the results is that open nesting and boosting do not perform well when the transaction size is small. This is because these transaction forms carry a certain amount of overhead—prominent at transaction size 1, for example. Much of this overhead is in acquiring abstract locks. Also, for each nested operation, the inner transaction (which is open) needs to create an abort handler and log it. These costs become smaller in a relative sense as transaction size increases, giving these forms better performance and scaling at larger transaction sizes.



(a) 0% updates (read-only)

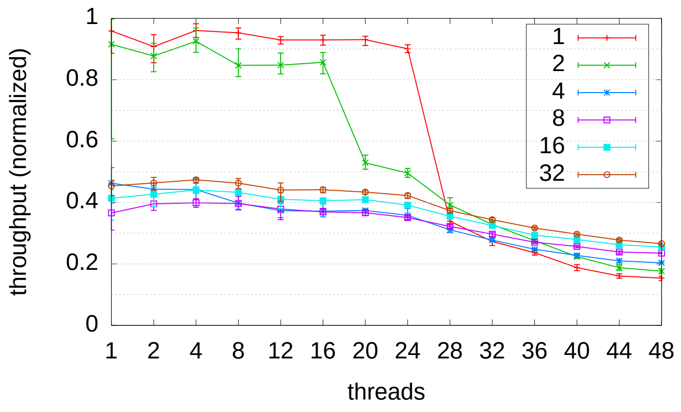


(b) 5% updates

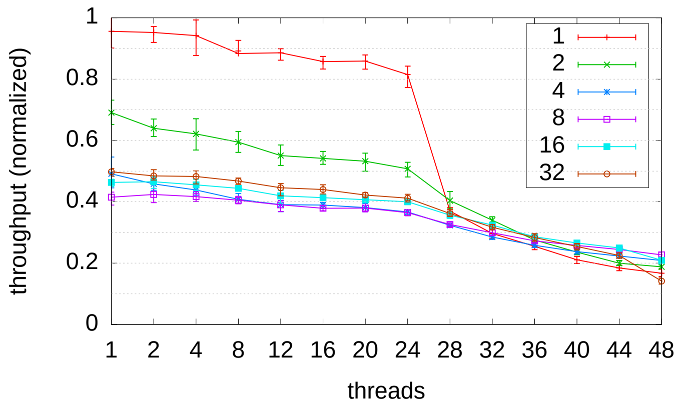


(c) 50% updates

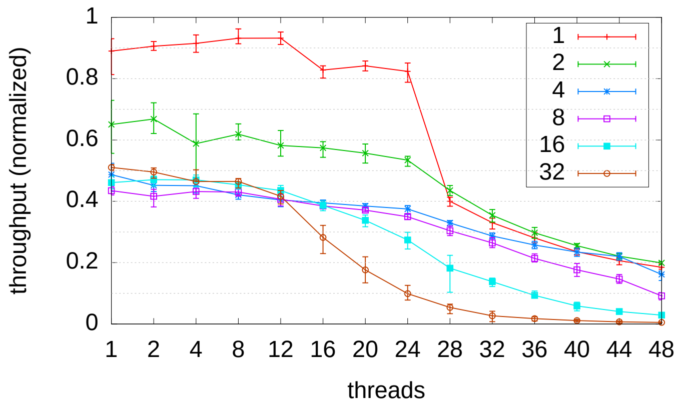
Figure 5.2.: Deuce (elastic), varying g



(a) 0% updates (read-only)

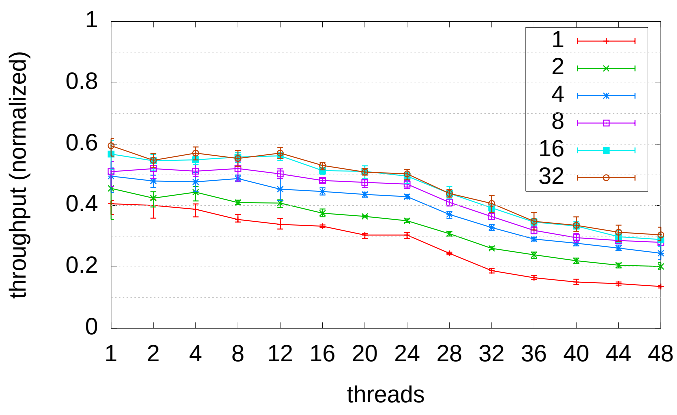


(b) 5% updates

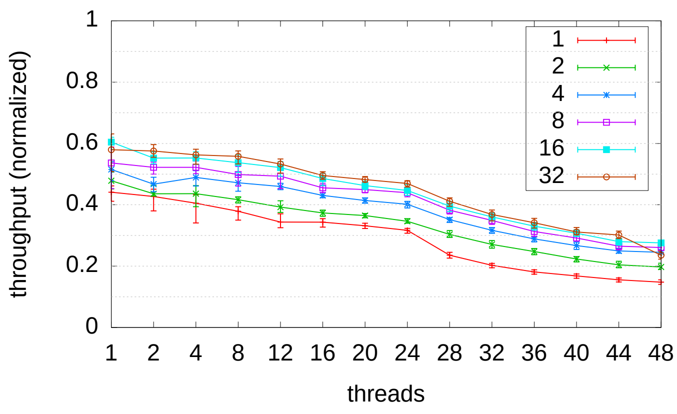


(c) 50% updates

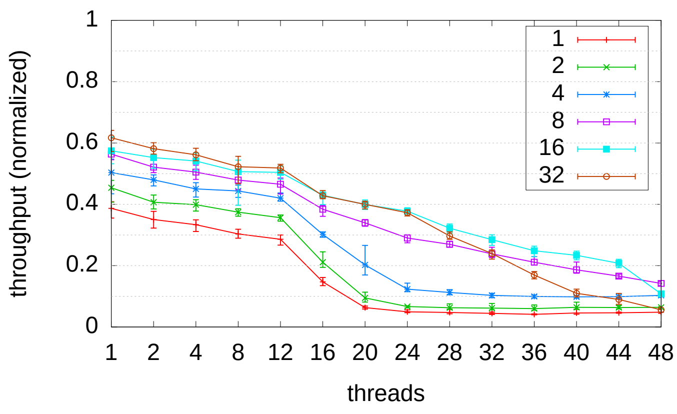
Figure 5.3.: Closed, varying g



(a) 0% updates (read-only)

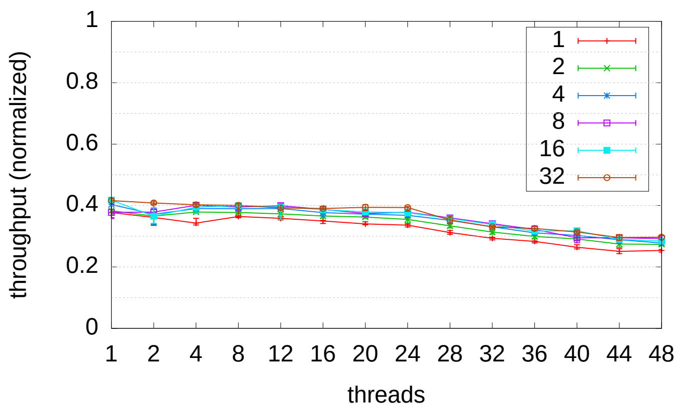


(b) 5% updates

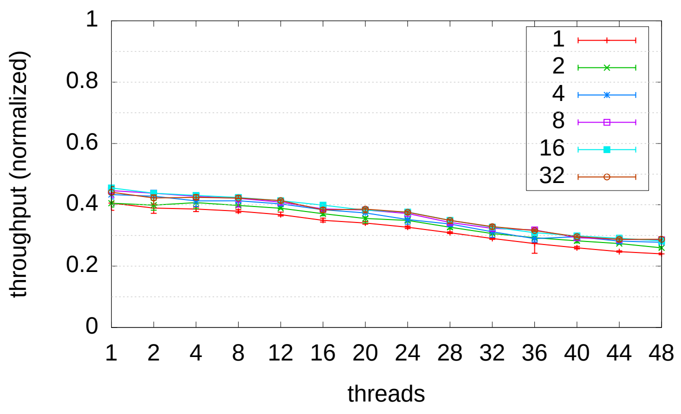


(c) 50% updates

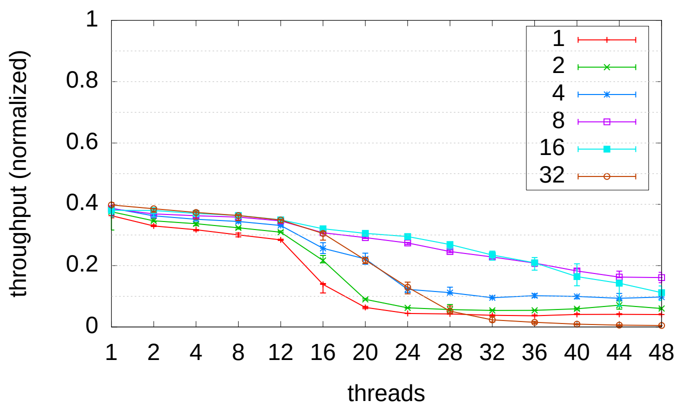
Figure 5.4.: Open, varying g



(a) 0% updates (read-only)



(b) 5% updates



(c) 50% updates

Figure 5.5.: Boosted, varying g

5.3.1 HTM versus STM

We first compare HTM and STM. Figure 5.1 shows three different transaction (group) sizes, 1, 2, and 4, from left to right. Within each group we have bars for each thread count (1, 2, ..., 48). The bars show the mean number of committed operations or hardware aborts per 5s benchmark iteration, breaking committed operations down by whether they ran in HTM or STM, with STM stacked on top of HTM. The left bar in each pair is for open nesting, the right bar for closed. Software aborts are so few as to be invisible in this graph. Finally, HTM abort counts are stacked on top (sometimes so few they are not visible). We connect HTM and total commits by lines, to help see the trends better. The bluer colors represent HTM, the yellower ones STM, and red represents aborts. These results are for update fraction 5%.

We find that open nesting performs relatively poorly due to the extra overhead of abstract locking and logging of undo operations, except at group size 4 where it outperforms closed nesting at all thread counts. This trend continues with higher group sizes (not shown). For thread counts beyond 24, threads start to share the same core (hyperthreading), which results in poorer performance, especially for HTM since a core's hyperthreads share L1 and L2 caches, which are used as the transactional buffer by TSX. This is exhibited by the drop in HTM commits and increase in HTM aborts. We also see that closed HTM falls away quickly as we increase the group size. Closed HTM largely fails beyond group size of 4. This is more because of transaction footprint exceeding the buffer than because of increasing conflicts. However, open HTM is strong in group size 4 and beyond. This is because the top-level transactions here are in software and each HTM transaction handles just one operation. This keeps the HTM footprint small while amortizing the open nesting overheads. Even with open nesting we see a relative increase in STM versus HTM beyond 24 threads, as a result of hyperthreading. The overall shape of the graphs for other update percentages are similar to these, and hence we do not show them.

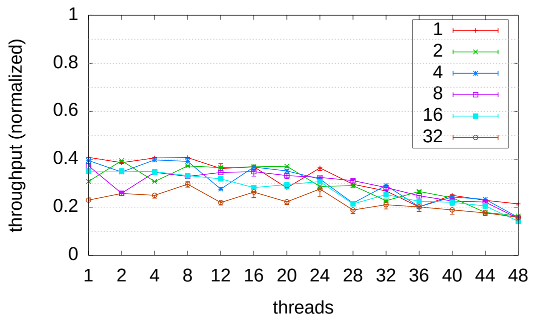
A theme here that we will see in other results as well is that there are portions of the parameter space where HTM works well and offers substantial speed up over STM (even with our hand optimization of STM). Likewise, there are portions of the space where open nesting works better than closed nesting, despite its higher overheads.

5.3.2 Closed, Open, and Boosted

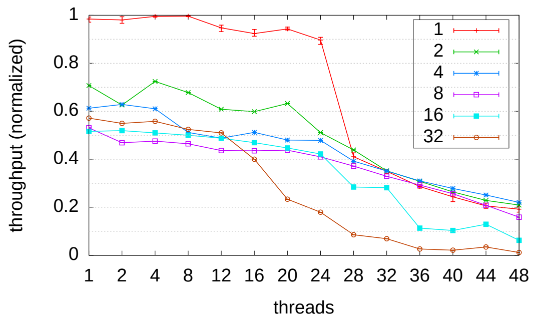
Figures 5.2 to 5.5 show normalized throughput for update fractions 0%, 5%, and 50%, respectively. Each figure includes four graphs, showing performance for Deuce [33] (running its efficient elastic mode transactions), closed nesting, open nesting, and boosting. We include Deuce since it demonstrates that our system lies in the same general performance range as this mature system. We see that closed nesting does better than Deuce at small thread counts and the same or not quite as well at large thread counts. We also see that for smaller thread counts and group sizes 1 and 2, closed nesting achieves particularly good performance. This is because those cases run in HTM much of the time. We compared open and closed nesting above and these graphs are consistent with that analysis. Boosting is interesting to compare with open nesting since a boosted data structure is hand crafted to offer good throughput for individual operations, and our wrappers implement the same abstract locking and undo logging for both boosting and open nesting. Being hand-crafted, we expected boosting to do better, but not surprisingly open nesting tends to win up to 12 threads where HTM remains effective.

5.3.3 Smaller Data Structure Size

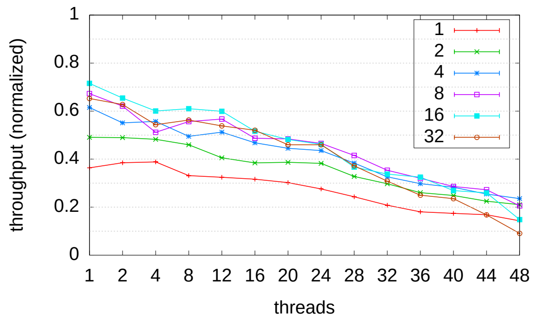
Figure 5.6 shows the impact due to increased chance of conflicts when using a smaller data structure, with 16K entries instead of 64K, key range of 32K, and update fraction 5%. For the same update fraction this smaller tree size results in more conflicts (both physical and abstract) than for larger trees, and the graphs clearly show how performance drops off with increasing group size since more transactions will conflict.



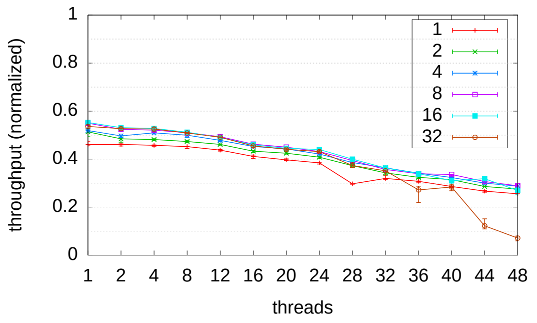
(a) Deuce (elastic)



(b) Closed



(c) Open



(d) Boosted

Figure 5.6.: 5% updates, varying g for tree size of 16K

6 FUTURE WORK

XJ relies on somewhat heavyweight mechanisms to support transactional memory mechanisms on a mostly unmodified OpenJDK platform. The bulk of the effort to make Java code run transactionally in XJ is achieved via byte-code and class rewriting at class load time. The only extension to OpenJDK is to allow injection of Intel's TSX hardware transactional memory (HTM) instructions into execution of interpreted and compiled code via HotSpot intrinsics. Two particular shortcomings of that approach are the addition of a transactional metadata word as an extra instance variable in all objects, and some jumping through hoops to convince the HotSpot optimizing compilers to compile HTM-enabled transactions. We discuss both of these issues with respect to OpenJDK and consider alternative implementations that represent a tighter integration with the OpenJDK implementation for improved performance.

6.1 Locking Protocol

XJ performs conflict detection at the level of objects, and tracks writes at the level of fields using an undo log. In the prototype STM implementation, each object carries an *extra* transactional metadata field, which holds the lock for writes, and otherwise contains a version number for the object, which is incremented upon commit. However, in XJ HTM and STM can safely co-exist and execute concurrently. Thus the two mechanisms need to play well with each other. In general, we adopt pessimistic concurrency control for writes, and optimistic concurrency control for reads. When running under STM, writes acquire a *write lock* on the object, which is noted in the metadata field—only one transaction can write to the object at a time. Readers proceed optimistically under STM, simply logging the value of the metadata field (a version number), and the log is then processed at commit time to validate the transaction (if the logged version number does not match the current value

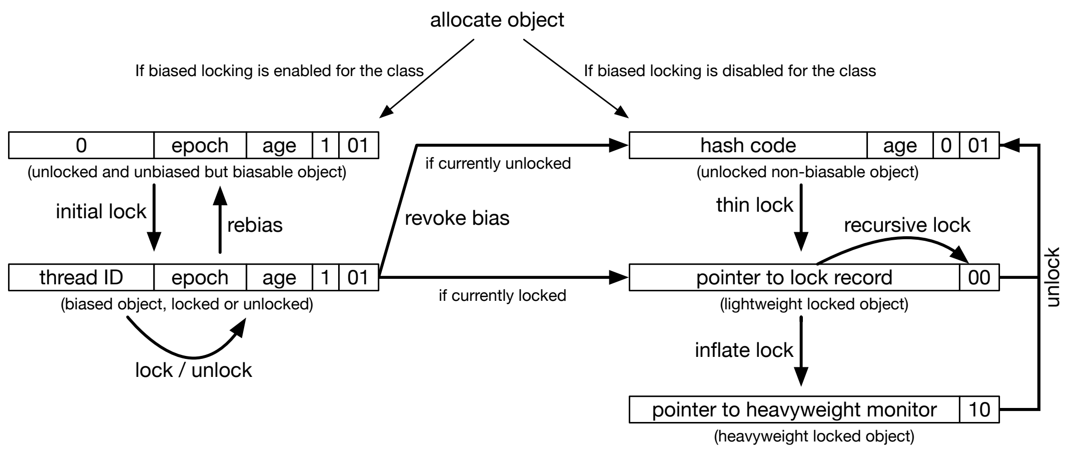

allocate object
If biased locking is enabled for the class
If biased locking is disabled for the class
0
epoch
age
1
01
(unlocked and unbiased but biasable object)
initial lock
rebias
thread ID
epoch
age
1
01
(biased object, locked or unlocked)
lock / unlock
if currently unlocked
revoke bias
if currently locked
hash code
age
0
01
(unlocked non-biasable object)
thin lock
recursive lock
pointer to lock record
00
(lightweight locked object)
inflate lock
pointer to heavyweight monitor
10
(heavyweight locked object)
unlock

Figure 6.1.: HotSpot standard synchronization (reproduction under GPLv2 license of a figure appearing in Kotzmann and Wimmer [34]).

and the owner of a locked object is not the current transaction then the transaction aborts). When running under HTM, writers commit by incrementing the version number, thus invalidating conflicting STM readers and conflicting with both HTM readers and writers. Reads under HTM perform a check to make sure that the object is not locked by another transaction, explicitly aborting if necessary. In sum, the lock/version word "glues" together the STM and HTM schemes into a coherent (and safe!) hybrid TM. We now present details on how this locking protocol can be integrated into OpenJDK, to be more efficient, rather than having to rely on an extra field added to each object.

6.1.1 Integrating Per-Object Transactional Metadata

The XJ prototype uses byte-code rewriting at load time to make every transactional application class inherit from a new `TransactionalObject` class, which has the transactional metadata word as its only instance field. Adding a field to each transactional object is costly in space and also in time to initialize and access the field. Ideally we would like this word to be a part of the object header. In OpenJDK every object is preceded by a class pointer (the "klass" word, which is native-sized or 32 bits depending on the use of compressed object pointers) and a header word. These are optionally followed by a 32-bit length word (if the object is an array), a 32-bit gap (if required by alignment rules), and then the object itself, comprising zero or more instance fields, array elements, or metadata fields. One option would have been to add another word to the header to store the transactional metadata. This would have worked well and is simple, but we want to do even better in terms of space and performance. Instead, we took a closer look at the format of the header word. Figure 6.1 shows the layout of the header word and how its contents evolve during the standard locking/unlocking process of Java object synchronization expressed using the `synchronized` keyword.

The most significant bits of the header word typically store multiple pieces of information as shown in Figure 6.1. These bits represent a hash code when the object is hashed, a thread id when the object is biased locked, a pointer to a lightweight lock, or a pointer to a heavyweight lock. The three lowest-order bits of the header word indicate which pieces of information the header holds. When an object is created and initialized it resides in the unlocked state (the most significant bits store no information). From this state an object can either transition to a hashed state or a biased locked state. If an object is hashed and a lock is requested (or vice versa), the object then transitions to a lightweight lock (the hash code and the thread id are moved into a lock record allocated on the stack). Lightweight locked objects that become subject to contention when another thread tries to lock them are "inflated": the object moves into a state where it refers to a heavyweight lock.

Transactional memory can be seen as an alternative method to achieve the same effect as `synchronized`: atomic updates to objects. It is reasonable to assume that any particular object is unlikely to be locked using both mechanisms, at least not at the same time. Thus an object that is participating in a transaction will not typically undergo all the states shown in Figure 6.1. We took this into account and tried to devise a mechanism to store the transactional meta-data in the existing header word. To do this we need a bit to indicate that the object is locked in transactional mode. We observed that we could accomplish this by enforcing 8-byte alignment on the "pointer to lock record" and the "pointer to heavyweight monitor". This gives us an extra bit to indicate that the object is being manipulated in

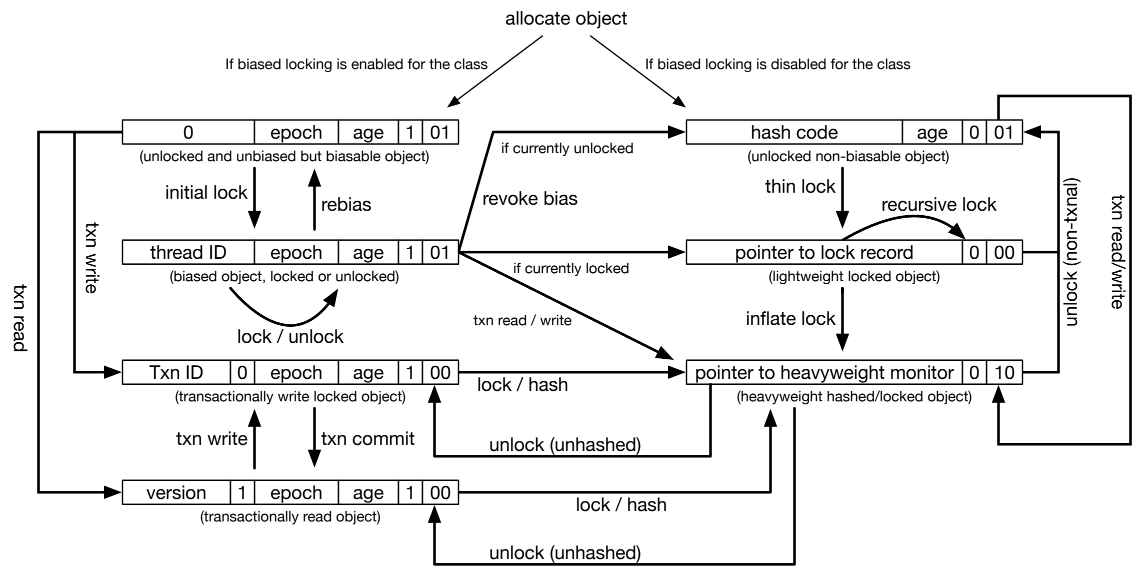

allocate object
If biased locking is enabled for the class
If biased locking is disabled for the class
0 | epoch | age | 1 | 01
(unlocked and unbiased but biasable object)
initial lock
rebias
thread ID | epoch | age | 1 | 01
(biased object, locked or unlocked)
lock / unlock
Txn ID | 0 | epoch | age | 1 | 00
(transactionally write locked object)
txn write
txn commit
version | 1 | epoch | age | 1 | 00
(transactionally read object)
txn write
txn read
if currently unlocked
revoke bias
if currently locked
txn read / write
lock / hash
unlock (unhashed)
lock / hash
unlock (unhashed)
hash code | age | 0 | 01
(unlocked non-biasable object)
thin lock
recursive lock
pointer to lock record | 0 | 00
(lightweight locked object)
inflate lock
pointer to heavyweight monitor | 0 | 10
(heavyweight hashed/locked object)
unlock (non-txnal)
txn read/write

Figure 6.2.: Proposed extension to the object header mark word

transactional mode. Figure 6.2 shows how the contents of the mark word evolve under this scheme.

The proposed scheme allows us to store the transactional meta-data in the existing object header word, as long as the object remains unhashed and is not `synchronized`. Our approach allows efficient access to the transactional meta-data for the object when it is stored in the header word. This is the most common case, and our proposed scheme is optimized for it. One bit of the transaction meta-data is used to indicate if the value stored is a transaction id (to indicate that the object is write locked) or its transactional version number. If a hash code is requested for a transactional object, or it becomes `synchronized`, then the transactional meta-data will be moved to a heavyweight monitor ("fat lock"). The monitor will be augmented with a field to be used as the lock/version for transactions. It need not incur all the overhead of a standard object monitor except when used (in the rare case) for both transactional access and `synchronized` manipulations. If an unhashed object is unlocked then the transactional meta-data will be moved back into the header word making it more efficient.

6.1.2 Handling Statics

Integrating the transactional metadata into the object header word solves the transactional locking issue for instance fields of an object, but it does not address static fields of a class. We need to handle statics separately. In the XJ prototype this was done by moving the static fields into a separate static singleton object, which allowed us to use the same locking scheme used for instance fields on the static fields. For our modified OpenJDK VM we propose to have a distinct static field (a synthetic field) to hold the transaction metadata for the static fields of the object. The proposed scheme can be extended to have a distinct lock word for disjoint subsets of the statics, if that added complexity offers enough performance advantage. This might increase concurrency and could be easily implemented via an annotation, similar to the existing `@Contended` annotation, on a group of static fields.

6.1.3 Handling Arrays

Using a single lock to protect a whole array does not scale in general since it will become a concurrency bottleneck. The XJ prototype injects wrapper classes at class load time for arrays, but we would prefer an integrated solution that allocates arrays as *arraylets*. These have been used to good advantage in real-time Java implementations [2; 47; 53; 54]. The size of these segments could be specified by the user. The integrated solution would allocate a transactional metadata word for each arraylet, solving the concurrency bottleneck issue for large arrays.

6.2 Interpreter and Compiler Concerns

As is well known, HotSpot has a byte-code interpreter as well as two levels of optimizing just-in-time (JIT) compilers (C1 and C2) that produce native code. Given the amount of work that the interpreter does, the data structures it touches and updates, etc., HTM will not work when interpreting byte-codes. This is because Intel's TSX hardware piggybacks on caching protocols and thus has a limited buffer size causing interpreted HTM transactions to fail due to buffer overflow. However, STM transactions can execute in the interpreter. HotSpot already uses reasonable heuristics to decide when it might be profitable to generate and execute native code. For transactional code, it might be useful to adjust those heuristics a bit since, once code is JIT compiled, HTM may be useful and HTM appears to run 5-10 times faster than STM for successful HTM transactions. But our main point is that HTM becomes interesting only for compiled code.

One of the main issues we encountered early on with using HTM was that many transactions failed with result code 0 (i.e., no specific reason given). Using the Intel Software Development Emulator, we found these aborts to be caused by execution of instructions that are incompatible with TSX—`FXRSTOR` and `FXSAVE` (perhaps among others)—and which are compiled into HotSpot's run-time stubs that control dynamic optimization and linking, and to resolve static and virtual method calls. By design, the HotSpot compilers patch these call sites at run time [43]. Thus our hardware transactions always failed, *and* those failures prevented triggering of the patching mechanism. Our workaround was to devise a mechanism to "warm" the system up in STM mode before attempting any hardware transactions. However, so that the compiler's optimizations will be triggered appropriately, and so that linking/patching will occur, these STM transactions had to follow the same code path (except for not using the TSX instructions) as HTM transactions did. We used a global flag to indicate whether we were in the software-only warm up phase, "weaving" together STM and HTM in the same code sequence, with if-then-else structure for each operation that HTM and STM handle differently.

This "weaving" strategy allowed us to executed HTM versions of methods in software to "snap links," etc., as we say. For example, a transaction might call some method `m` of the application where `m` is not yet JIT compiled. The HotSpot JIT compilers will insert a call to a *stub* routine that triggers compilation of the target method `m` if it is called, or, if by that time `m` has been compiled, will patch the stub to call the compiled code for `m`. Both behaviors of a stub cause an HTM transaction to fail, and unwind, thus not actually triggering the compilation or link-snapping behavior. We thus needed a way to execute the *same* *stub* under STM. Once the stub's behavior had been appropriately triggered, HTM would no longer fail going through that code path. The stubs of which we speak are examples of *guards*. We say a guard *succeeds* if it follows a path where no special condition needs fixing up; this will be a fast path. We say a guard *fails* when it follows a path for a fix up; this will be a slow path.

Weaving together HTM and STM versions leads to code that is probably slower than it can be, because of all the extra if-then-else blocks. Granted, good branch prediction reduces their cost some, but they still need to be executed and they may stress the branch predictor. It would be better to generate HTM code without these branches. We propose two ways to do this: (*i*) returning some information from a failing HTM transaction, and (*ii*) maintaining correlated HTM and STM versions of the code.

6.2.1 Using HTM Failure Codes

As previously mentioned, failing guards will cause HTM transactions to fail. This has the side-effect that the run-time system then does not know a guard failed and thus cannot fix it up. However, it turns out that an *explicit* abort of an Intel HTM transaction with the `XABORT` instruction can pass 8 bits of information back in the `EAX` register.[1] So, if a piece of code running under HTM has no more than 256 guards, the compiler could use the 8-bit code in the "XABORT" instruction to indicate which guard failed. This may allow a future execution of the transaction to succeed. (We say “may” because a future execution is not guaranteed to follow the same path through the code.)

But what if the HTM code region has more than 256 guards? This could happen in the presence of calls, etc. Here is a scheme to exploit multiple transaction attempts to extract more bits from the failing transactions and narrow down the set of failing guards to the one on which the system should act. First, we assume that there is a per-thread location (it could even be a register) that will indicate which retry of an HTM transaction with a failing guard

[1]As an aside to designers of future hardware, we observe that it appears useful to be able to return more bits, and possibly even to have a memory region not subject to HTM semantics in which one could store “side” results of a failing transaction.

we are on, and some previously returned information. The attempt number will initially be 0, will be 1 on the first retry, etc. The essence of the scheme is this. We assign each guard a unique number. We develop k hash functions (k is likely 4, given the particulars of our scheme), h_0 through h_{k-1}. On attempt j of a failing guard in a hardware transaction, we return $h_j(i)$ where i is the unique id of the failing guard. These hash function return a seven bit value. The eighth bit we use to indicate whether we are continuing or starting over. On attempts after the first, we check a failing guard's previous hash values against those noted as being returned by previous attempts. If they match, we indicate that and return the hash value for the current attempt. If they don't match, we indicate that and return $h_0(i)$. If we get through four attempts with matches, we will have 28 bits to identify a particular guard. In many cases we might need even fewer, but the scheme generalizes to extract any number of bits, at the cost of additional retries and the increasing risk that we may go down a different code path (depending on the nature of the transaction and of the guard). Notice that the hash codes can simply be groups of seven bits from the guard's unique number, which probably makes for simple code.

This scheme assumes that all we need to know is which guard failed. When updating a polymorphic inline cache (for example), we may desire to know which class was presented that was not in the cache. The same approach can be taken to extract more bits. An alternative would be to have code that would figure out which object's class was being dispatched on, etc. This could get complicated, so returning the information directly (if incrementally) may be simpler. It is certainly more general.

6.2.2 Maintaining Correlated Code Versions

An alternative to using the HTM failure codes is to maintain STM and HTM versions that have the same guards. This is like taking XJ's code and pulling out a version with all the "then" clauses of the HTM-STM if-then-else blocks, and another version with all the "else" clauses. Whenever an action is taken on a guard in one version of the code, we force the same action to occur on the other version. Thus, if the HTM code fails in a guard—a fact that can be indicated with just one distinguished result code value—we can run the STM version and if a guard fails, it will be fixed in both versions and we can try HTM the next time. If the HTM version can usefully indicate which guard failed (i.e., there are not too many guards, and the particular one does not require additional information), then the result code can be used to fix the guard in both versions and HTM retried. However, handling a failing guard is probably quite costly compared with the work that can succeed in an HTM transaction, and even compared with an STM version of that same work, so always running the STM version to trigger guard fixing is a reasonable strategy.

6.2.3 Further Optimizations

In XJ we supported hand annotation of various actions in a transaction, to enable us to elide locking or logging work. This is particularly applicable to STM code, since HTM inherently avoids some of the work, but it is also relevant to HTM code. We envisioned a byte-code optimizer that would perform the needed data flow analyses and then rewrite the byte-code (or insert the annotations). This could be done as an additional optimization pass in HotSpot, particular to transaction code.

7 SUMMARY

The XJ language provides full blown support for closed, open nested and boosted transactions. The hybrid transactional memory system supported by the XJ framework allows hardware transactions and software transactions to proceed concurrently. Open nesting increases the envelope of concurrency and transaction sizes that can be accommodated in hardware.

In this dissertation we have presented XJ, a dialect of Java supporting a range of transactional programming abstractions, including open/closed nested transactions, and transactional boosting. We also show how HTM can be used with nested transactions to boost performance. Additionally we've shown how STM and HTM can coexist with each other in the face of nesting, and show how the transition from HTM to STM and vice versa can be done automatically.

7.1 Conclusions

Our results demonstrate the utility of nesting as a means to achieving reliably scalable concurrent manipulation of data structures using open/closed nesting, without the need for hand-tuned and hand-coded non-blocking implementations. So long as the underlying data structure is friendly to transactions it can easily be nested.

Moreover, we demonstrate that HTM mechanisms can be exploited effectively to accelerate nested transaction schemes, while allowing software-only schemes to run safely alongside the HTM-accelerated executions.

Our results indicate the degree to which hyperthreading degrades performance of HTM schemes due to the need to share capacity between hyperthreads on the same core.

We also demonstrate the performance envelopes for each of the schemes, showing that there is a space in the workload spectrum where each is superior. As such, programmers must choose carefully which technique to employ, depending on the nature of their programs.

For programmers willing to wrap bespoke linearizable data structures, boosting works well at high thread counts where HTM degrades, because it does not pay the performance penalty of STM.

We have also shown how to integrate HTM features into OpenJDK such that the compilers can inline the HTM operations as intrinsics. In future work we plan to convince the Hotspot compilers to warm up more effectively and optimize the HTM code.

Finally, We also show how the HotSpot optimizing compilers can be modified to be aware of transactions, such that HTM can be used in production. We propose two complementary modifications to the compiler that avoid having to warm the system up prior to using HTM. We also discuss other optimizations that the compiler could perform on HTM methods.

LIST OF REFERENCES

LIST OF REFERENCES

[1] C. S. Ananian, K. Asanovic, 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, pages 316–327, San Francisco, California, USA, 2005. ISBN 0-7695-2275-0. doi: 10.1109/HPCA.2005.41.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 285–298, New Orleans, Louisiana, USA, 2003. ISBN 1-58113-628-5. doi: 10.1145/604131.604155.

[3] E. Bruneton. *ASM 4.0: A Java bytecode engineering library*, Sept. 2011. URL `http://download.forge.objectweb.org/asm/asm4-guide.pdf`. Version 2.0.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 1–13, Ottawa, Ontario, Canada, 2006. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133983.

[5] K. Chapman, A. L. Hosking, J. E. B. Moss, and T. Richards. Closed and open nested atomic actions for Java: Language design and prototype implementation. In *International Conference on the Principles and Practice of Programming on the Java platform: virtual machines, languages, and tools*, PPPJ, pages 169–180, Cracow, Poland, Sept. 2014. doi: 10.1145/2647508.2647525.

[6] K. Chapman, A. L. Hosking, and J. E. B. Moss. Hybrid STM/HTM for nested transactions on OpenJDK. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 569–585, Amsterdam, The Netherlands, Oct. 2016. doi: 10.1145/2983990.2984029. Distinguished Paper Award.

[7] K. Chapman, A. L. Hosking, and J. E. B. Moss. Extending OpenJDK to support hybrid STM/HTM: Preliminary design. In *ACM SIGPLAN Workshop on Virtual Machines and Intermediate Languages*, VMIL, Amsterdam, The Netherlands, Oct. 2016. doi: 10.1145/2998415.2998417.

[8] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, and H. Z. M. Tremblay. Rock: A high-performance Sparc CMT processor. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, volume 29 of *MICRO*, pages 6–16, March 2009. ISBN 0272-1732. doi: 10.1109/MM.2009.34.

[9] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous speculative threading: A novel pipeline architecture implemented in Sun's Rock processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA, pages 484–495, Austin, TX, USA, 2009. ISBN 978-1-60558-526-0. doi: 10.1145/1555754.1555814.

[10] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 extension for lock-free data structures and transactional memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 39–50, 2010. ISBN 978-0-7695-4299-7. doi: 10.1109/MICRO.2010.40.

[11] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly methodology for search structures. Research report, INRIA, Feb. 2012. URL https://hal.inria.fr/hal-00668010.

[12] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 161–170, New Orleans, Louisiana, USA, 2012. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145837.

[13] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 67–78, Bangalore, India, 2010. ISBN 978-1-60558-877-3. doi: 10.1145/1693453.1693464.

[14] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 39–52, Newport Beach, California, USA, 2011. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950373.

[15] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 336–346, San Jose, California, USA, 2006. ISBN 1-59593-451-0. doi: 10.1145/1168857.1168900.

[16] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 157–168, Washington, DC, USA, 2009. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508263.

[17] D. Dice, A. Kogan, and Y. Lev. Refined transactional lock elision. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 1–19, Barcelona, Spain, 2016. ISBN 978-1-4503-4092-2. doi: 10.1145/2851141.2851162.

[18] N. Diegues and P. Romano. Self-tuning Intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing*, ICAC, pages 209–219, Philadelphia, PA, USA, June 2014. ISBN 978-1-931971-11-9.

[19] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT, pages 3–14, Edmonton, AB, Canada, 2014. ISBN 978-1-4503-2809-8. doi: 10.1145/2628071.2628080.

[20] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC, pages 131–140, Zurich, Switzerland, 2010. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835736.

[21] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 1–10, San Francisco, CA, USA, 2015. ISBN 978-1-4503-3205-7. doi: 10.1145/2688500.2688501.

[22] 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 Annual International Symposium on Computer Architecture*, ISCA, page 102, M&#252;nchen, Germany, 2004. ISBN 0-7695-2143-6.

[23] T. L. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA, pages 388–402, Anaheim, California, USA, 2003. ISBN 1-58113-712-5. doi: 10.1145/949305.949340.

[24] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 207–216, Salt Lake City, UT, USA, 2008. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345237.

[25] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA, pages 289–300, San Diego, California, USA, 1993. ISBN 0-8186-3810-9. doi: 10.1145/165123.165164.

[26] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, PODC, pages 92–101, Boston, Massachusetts, USA, 2003. ISBN 1-58113-708-7. doi: 10.1145/872035.872048.

[27] M. P. Herlihy and J. M. Wing. Linearizability: A correctness criterion for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972.

[28] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC, pages 82–91, San Jose, California, USA, 2006. ISBN 1-59593-578-9. doi: 10.1145/1178597.1178611.

[29] Intel. *Intel Transactional Synchronization Extensions (Intel TSX) programming considerations*. URL https://software.intel.com/en-us/node/524023.

[30] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System Z. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 25–36, Vancouver, B.C., CANADA, 2012. ISBN 978-0-7695-4924-8. doi: 10.1109/MICRO.2012.12.

[31] E. H. Jensen, G. W.Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Nov 1987. URL https://e-reports-ext.llnl.gov/pdf/212157.pdf.

[32] A. Kasko, S. Kobylyanskiy, and A. Mironchenko. *OpenJDK Cookbook*. Packt Publishing, Jan. 2015. ISBN 1849698406.

[33] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Workshop on Programmability Issues for Heterogeneous Multicores*, MULTIPROG, Pisa, Italy, Jan. 2010.

[34] T. Kotzmann and C. Wimmer. *OpenJDK Wiki: Synchronization and Object Locking*, 2008. URL https://wiki.openjdk.java.net/display/HotSpot/Synchronization.

[35] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 209–220, New York, New York, USA, 2006. ISBN 1-59593-189-9. doi: 10.1145/1122971.1123003.

[36] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Transact, Aug. 2007. URL http://hdl.handle.net/1802/4431.

[37] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2004.

[38] A. Matveev and N. Shavit. Reduced hardware NORec: A safe and scalable hybrid transactional memory. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 59–71, Istanbul, Turkey, 2015. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694393.

[39] P. McGachey, A. L. Hosking, and J. E. B. Moss. Class transformations for transparent distribution of Java applications. *Journal of Object Technology*, 10:9:1–35, 2011. ISSN 1660-1769. doi: 10.5381/jot.2011.10.1.a9.

[40] J. E. B. Moss. *Nested Transactions: An approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1981.

[41] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, Dec. 2006. ISSN 0167-6423. doi: 10.1016/j.scico.2006.05.010.

[42] Y. Ni, V. 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 *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 68–78, San Jose, California, USA, 2007. ISBN 978-1-59593-602-8. doi: 10.1145/1229428.1229442.

[43] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium — Volume 1*, JVM, page 1, Monterey, California, USA, 2001.

[44] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979. ISSN 0004-5411. doi: 10.1145/322154.322158.

[45] M. Payer and T. R. Gross. adaptSTM: An online fine-grained adaptive STM system. Technical report, ETH Zurich, 2010.

[46] M. Payer and T. R. Gross. Performance evaluation of adaptivity in software transactional memory. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 165–174, April 2011. doi: 10.1109/ISPASS.2011.5762733.

[47] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 146–159, Toronto, Ontario, Canada, 2010. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806615.

[48] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO, pages 294–305, Austin, Texas, USA, 2001. ISBN 0-7695-1369-7.

[49] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 5–17, San Jose, California, USA, 2002. ISBN 1-58113-574-2. doi: 10.1145/605397.605399.

[50] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA, pages 494–505, 2005. ISBN 0-7695-2270-X. doi: 10.1109/ISCA.2005.54.

[51] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 53–64, San Jose, California, USA, 2011. ISBN 978-1-4503-0743-7. doi: 10.1145/1989493.1989501.

[52] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 187–197, New York, New York, USA, 2006. ISBN 1-59593-189-9. doi: 10.1145/1122971.1123001.

[53] J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, and K. S. McKinley. Z-rays: Divide arrays and conquer speed and flexibility. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 471–482, Toronto, Ontario, Canada, 2010. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806649.

[54] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES, pages 9–17, San Jose, California, USA, 2000. ISBN 1-58113-338-3. doi: 10.1145/354880.354883.

[55] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, Nov. 1993. ISSN 1063-6552. doi: 10.1109/88.260295.

[56] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–19, Denver, Colorado, USA, 2013. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503232.

[57] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 23–34, Los Alamitos, CA, USA, Apr. 2007. doi: 10.1109/ISPASS.2007.363733.

[58] L. Zhang. UVSIM reference manual. Technical report, University of Utah, Mar. 2003. UUCS-03-011.

VITA

VITA

Keith Godwin Chapman was born in Colombo, Sri Lanka. After completing his primary and secondary education at Wesley College in Colombo, Sri Lanka, Keith obtained admission to the Engineering faculty at the University of Moratuwa, Sri Lanka. Upon receiving his Bachelor of Science in Computer Science & Engineering, Keith joined WSO2 Inc, in August 2006 as a software engineer. At WSO2, he rose through the ranks rapidly, been promoted to senior software engineer, technical lead and product manager. In August 2009, he began his graduate studies at Purdue University, West Lafayette, Indiana, USA, under the direction of Professor Antony Hosking. Keith, earned his Master of Science in Computer Science from Purdue University, West Lafayette, Indiana, USA in May 2012. His research interests are in the areas of programming language design and implementation, compilers, language runtimes, virtual machines, garbage collection, concurrency and transactional memory. During his Ph.D. studies at Purdue University, Keith interned at IBM T. J. Watson Research Center, Yorktown Heights, New York, USA, Microsoft Research, Redmond, Washington, USA and IBM Research in Austin, Texas, USA.