Fast Software Transactions

by

Michael F. Spear

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Dr. Michael L. Scott

Department of Computer Science
Arts, Sciences and Engineering
School of Arts and Sciences

University of Rochester
Rochester, New York

2009
To my wife, Emily.
Curriculum Vitae

Michael Fuchs Spear (né Michael Vincent Fuchs) was born in Suffern, New York on November 1, 1977. He attended the United States Military Academy at West Point from 1995 to 1999, receiving a Bachelor of Computer Science degree in 1999. Upon graduation, he entered the United States Air Force as a Second Lieutenant. He was assigned to Elmendorf Air Force Base, Alaska, where he served as the Deputy Chief of Wing Programs for the 3rd Wing. While stationed at Elmendorf, he earned his Masters of Business Administration from the University of Alaska, Anchorage. In 2002, he became the Executive Officer of the 314th Maintenance Group, Little Rock Air Force Base, Arkansas. In 2003, he was promoted to the rank of Captain, which he held until separating from the Air Force in 2004. In the fall of 2004, he joined the Ph.D. program of the Department of Computer Science at the University of Rochester, where he worked under the guidance of Professor Michael L. Scott. In 2006 he received a Masters of Science degree from the University of Rochester. He interned at Microsoft Research in 2005, and at the IBM T. J. Watson Research Center in 2007.
Acknowledgments

First and foremost, I would like to thank Professor Michael L. Scott. He has been an outstanding advisor, collaborator, and role model. Without his guidance and his trust, many of my proudest accomplishments during these past five years would never have come to be.

I would also like to thank my thesis committee members: Sandhya Dwarkadas, Chen Ding, and Michael Huang. Their shrewd and insightful reviews have undoubtedly improved the quality of my research, and they have helped me to grow as a scholar.

During my internship at Microsoft Research, I had the privilege of working with Galen Hunt. I had no research experience when I arrived at Microsoft. Galen saw this as an opportunity to shape my research skills and approach. His discipline and dedication are a continuing source of inspiration.

I am deeply grateful to Maged Michael, Christoph von Praun, and Peng Wu. Chapters 3, 4, and 6 stem directly from our collaboration, both during my internship at IBM and afterward. Their demand for rigor and precision has had a lasting impact on my research. I would especially like to thank Maged, who was my mentor at IBM.

At Rochester, Bill Scherer and Virendra Marathe introduced me to Transactional Memory. Had they not made synchronization so approachable, and collaboration so enjoyable, I may never have pursued research in this area. It has been my good fortune
to have Arrvindh Shriraman and Luke Dalessandro nearby as well. When we have disagreed, it has almost always led to an exciting new idea. Jim Roche has provided outstanding equipment and software support, and has always been willing to adjust his schedule to my requests.

Throughout my life, I have learned from excellent teachers. My parents, Jim and Kathy Fuchs, taught me to love learning. Richard Malone and Marty Manson encouraged me to stand up for what I believed in, even when it meant standing alone. Ingrid Arlt, Jean Blair, Edward Sobiesk, and Tommy Wagner filled me with a passion for computer science. Randall Harvey taught me how to lead, how to follow, and when each was appropriate. To them, and to the hundreds of other teachers in my life, I will always be grateful.

For the last five years, my family has been more supportive, loving, and forgiving than I ever thought possible. My children have been a constant source of joy, and were always understanding when a new idea or solution to a nagging problem arrived in the middle of our time together. Words cannot express my gratitude to my wife, Emily. She is an inspiring teacher, a loving parent, and a true friend. Without her by my side, this work would never have been possible. I am thankful for her dedication to me, and I dedicate this dissertation to her.

The work presented in this dissertation was supported by grants from the United States National Science Foundation (grants numbers: CSR-0720796, CCF-0702505, CNS-0615139, CNS-0411127, CCR-0204344, and CNS-0509270), equipment grants from Sun Microsystems and IBM, an IBM Faculty Partnership Award, and financial support from Sun Microsystems, Intel, and Microsoft. Any opinions, findings, and conclusions or recommendations presented in this work are those of the author(s) and do not necessarily reflect the views of the above named organizations.
Abstract

In the past, only a small group of highly-skilled programmers were expected to write programs that used multiple processors simultaneously. However, microprocessor vendors have recently turned to multi-core chip designs as the most profitable way to increase performance. We are now seeing multi-core processors in desktops, laptops, handheld computers, and even embedded devices. As a result, parallel programming is becoming a core competency for all programmers.

When concurrent threads of a parallel program share data, they must ensure that certain sets of accesses to shared memory execute as indivisible operations (that is, the regions must appear to execute atomically). The dominant mechanism for providing atomicity is mutual exclusion locks. Unfortunately, locks are a very low-level mechanism, widely regarded as too difficult for the average programmer to use correctly. Fortunately, locks are not the only mechanism that provides atomicity. Database transactions (used by millions of programmers to write highly concurrent e-commerce code) also ensure that regions of code execute atomically. Many efforts are underway to employ transactions to implement atomicity within general-purpose programming languages, in the form of Transactional Memory (TM).

TM can be implemented in hardware, software, or a combination of the two. Pure software TM (STM) systems are of particular interest, since they can serve as a replacement for lock-based atomicity in millions of existing multicore processors. In order for
an STM implementation to provide a viable implementation of atomicity, it must not introduce substantial single-thread latency, it must offer good scalability and throughput, and it must be general enough to serve as a replacement for most, if not all, of the uses of locks.

This dissertation introduces three techniques to reduce the latency of STM. The first source of latency we target relates to consistency checks. While some amount of checking is necessary by an STM runtime to detect when two in-flight transactions wish to make incompatible accesses to the same region of memory, we observe that a lightweight *commit counter* can avoid most checks, without weakening the STM’s correctness guarantees. Secondly, we present the RingSTM algorithm, which further lowers overheads by decreasing the precision of conflict detection: by summarizing the access patterns of transactions with fixed-size Bloom filters, RingSTM is able to avoid most uses of costly read-modify-write operations. Lastly, we show how STM-specific compiler optimizations can decrease the cost of memory fences for STM algorithms implemented on processors with relaxed memory consistency.

Some of our latency-reducing mechanisms also improve throughput. The commit counter heuristic can approximate a novel conflict-detection scheme called *mixed invalidation*, which prevents transactions from aborting due to conflicts between reader and writer transactions, so long as the reader commits first. The RingSTM algorithm guarantees progress, ensuring that no transaction aborts unless some other transaction commits. We also present a novel approach to contention management (the mechanisms used by an STM to ensure forward progress). Our contention management policy ensures good throughput even with high rates of conflict. It also provides a degree of fairness under high contention, without introducing latency when contention is low.
Locks are often used to protect I/O operations, and lock-based code regularly co-exists with unsynchronized code. Since most STM implementations use speculation and rollback, they cannot easily support I/O and irreversible operations unless they prevent all concurrency whenever a transaction wishes to perform an irreversible operation. We show how to allow an inevitable transaction (one that will not roll back, and thus can perform irreversible operations) to execute concurrently with non-inevitable transactions. We also present a semantics for STM based on strict serializability. This semantics allows transactions to transition data between shared and thread-local states in the same manner as locks. Furthermore, our semantics preserves serializability of all transactions, without introducing latency at the boundaries of transactions that do not transition data between shared and thread-local states. Taken together, our inevitability mechanisms and semantics greatly improve the generality of transactional memory.
Table of Contents

Curriculum Vitae iii

Acknowledgments iv

Abstract vi

List of Tables xvii

List of Figures xviii

List of Algorithms xx

1 Introduction 1

1.1 The Synchronization Problem 3

1.1.1 Hardware Instructions for Synchronization 5

1.1.2 Locking and Nonblocking Synchronization 7

1.1.3 Problems With Locks 8

1.1.4 Composability 10

1.2 The Case for Atomicity 11
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3</td>
<td>Atomicity Through Transactional Memory</td>
<td>13</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Abstract STM Execution</td>
<td>14</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Example Implementation</td>
<td>15</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Implementation Alternatives</td>
<td>21</td>
</tr>
<tr>
<td>1.4</td>
<td>Alternatives to Transactional Memory</td>
<td>23</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Speculative Lock Elision</td>
<td>23</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Speculative Parallelization of Sequential Code</td>
<td>24</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Summary</td>
<td>25</td>
</tr>
<tr>
<td>1.5</td>
<td>Problems With STM Implementations</td>
<td>26</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Single-Thread Latency</td>
<td>26</td>
</tr>
<tr>
<td>1.5.2</td>
<td>Throughput and Generality</td>
<td>27</td>
</tr>
<tr>
<td>1.5.3</td>
<td>Interaction with Nontransactional Code</td>
<td>28</td>
</tr>
<tr>
<td>1.6</td>
<td>Contributions</td>
<td>31</td>
</tr>
<tr>
<td>1.6.1</td>
<td>Decreasing Latency</td>
<td>31</td>
</tr>
<tr>
<td>1.6.2</td>
<td>Increasing Throughput</td>
<td>32</td>
</tr>
<tr>
<td>1.6.3</td>
<td>Increasing Generality Without Sacrificing Performance</td>
<td>33</td>
</tr>
<tr>
<td>1.7</td>
<td>Roadmap</td>
<td>34</td>
</tr>
<tr>
<td>2</td>
<td>Safely Reducing The Cost of Conflict Detection</td>
<td>35</td>
</tr>
<tr>
<td>2.1</td>
<td>Techniques For Detecting and Resolving Conflicts</td>
<td>36</td>
</tr>
<tr>
<td>2.2</td>
<td>Overview of RSTM</td>
<td>41</td>
</tr>
<tr>
<td>2.3</td>
<td>The Global Commit Counter Heuristic</td>
<td>44</td>
</tr>
<tr>
<td>2.4</td>
<td>Experimental Evaluation of Conflict Detection and Validation Strategies</td>
<td>48</td>
</tr>
</tbody>
</table>
2.4.1 Strategies Considered ........................................ 49
2.4.2 Benchmarks ................................................. 50
2.4.3 Discussion of Results ........................................ 51

2.5 Related Work .................................................. 56
2.6 Conclusions .................................................... 61

3 Lowering The Latency Of Software Transactional Memory 63

3.1 Costs and Benefits of Orec STM .............................. 64
  3.1.1 Tradeoffs in Orec-Based STM ............................ 64
  3.1.2 Alternative Hardware and Software TMs ................. 66
  3.1.3 Contributions ............................................. 67

3.2 RingSTM Design ................................................. 69
  3.2.1 Global Metadata Describes Committed Transactions .... 69
  3.2.2 Trading Precision for Speed in Validation ............... 71
  3.2.3 Low Commit Overhead and Livelock Freedom ........... 72
  3.2.4 Contention Management ................................... 73
  3.2.5 Privatization is Free ..................................... 75
  3.2.6 Minimal Memory Management Overhead ................. 76

3.3 Implementation ................................................ 77
  3.3.1 The Default RingSTM Algorithm ......................... 78
  3.3.2 The Single Writer Algorithm (RingSW) ................. 81
  3.3.3 Relaxed Commit Order (RingR) ......................... 83
  3.3.4 Ring Rollover .......................................... 85
4.4.1 Removing Fences Within a Basic Block ........................................ 110
4.4.2 Tight Loop Optimizations ..................................................... 111
4.4.3 Removing Final Postvalidation ............................................... 113
4.4.4 Dynamically Checked, Unsafe Use .......................................... 113
4.4.5 Speculative Read Hoisting .................................................... 115

4.5 Experimental Evaluation ......................................................... 117
4.5.1 Optimization Levels ............................................................ 118
4.5.2 Analysis ............................................................................. 119

4.6 Related Work ......................................................................... 126

4.7 Conclusions ............................................................................ 127

5 Contention Management ............................................................. 129
5.1 Contention Management in Software Transactional Memory ........ 130
5.2 Baseline Lazy System ............................................................... 133
5.2.1 Write Set Lookup Latency .................................................. 134
5.2.2 Timestamps and Wasted Work ........................................... 136
5.2.3 Preventing Aborts ............................................................... 138
5.2.4 Approaching Livelock Freedom .......................................... 140

5.3 Defining Priority ..................................................................... 145
5.3.1 Fairness ............................................................................. 145
5.3.2 Supporting Inevitability ....................................................... 147
5.3.3 Integrating Retry ............................................................... 148
5.3.4 Automatic Priority Elevation .............................................. 149
5.4 Implementation .......................................................... 150
  5.4.1 Priority Read Bits .................................................. 153
  5.4.2 Priority Read Filters ............................................. 155
5.5 Evaluating Priority Mechanisms ................................. 157
  5.5.1 Breaking Write-Write Starvation .............................. 157
  5.5.2 Read-Write Starvation .......................................... 160
5.6 Summary ............................................................. 163

6 Inevitability Mechanisms for Software Transactional Memory 165
  6.1 Introduction .......................................................... 166
    6.1.1 Inevitability in Existing TM Systems ...................... 167
    6.1.2 Contributions ................................................ 168
  6.2 STM Support for Inevitability ................................. 168
  6.3 Inevitability Mechanisms ....................................... 171
    6.3.1 Global Write Lock ........................................... 172
    6.3.2 GWL + Transactional Fence ............................... 173
    6.3.3 Writer Drain ................................................ 174
    6.3.4 Inevitable Read Locks ...................................... 177
    6.3.5 Inevitable Read Filter .................................... 178
    6.3.6 Summary ................................................... 180
  6.4 Programming with Inevitability ............................... 181
    6.4.1 Irreversible Operations ................................... 181
    6.4.2 Non-Transactional Code and System Calls .............. 184
6.4.3 Condition Synchronization .................................. 186
6.5 Evaluation ......................................................... 187
  6.5.1 Latency for Non-Inevitable Transactions ................. 188
  6.5.2 Supporting Disjoint Writes ................................ 189
  6.5.3 Workload Acceleration .................................... 192
  6.5.4 Dependence on Workload Characteristics ............... 194
6.6 Conclusions ..................................................... 196

7 Ordering-Based Semantics for Software Transactional Memory 198
  7.1 Introduction .................................................. 199
  7.2 Background .................................................. 201
  7.3 Ordering-Based TM Semantics ............................... 206
    7.3.1 Strict Serializability .................................. 206
    7.3.2 Values read ............................................. 208
    7.3.3 Selective Strictness .................................... 209
    7.3.4 Asymmetric Flow ....................................... 211
  7.4 Implementing SSS and SFS Systems ........................ 213
    7.4.1 Preventing the Doomed Transaction Problem .......... 215
    7.4.2 Preventing the Delayed Cleanup Problem ............. 217
    7.4.3 Preventing Publication Errors ........................ 220
    7.4.4 Preventing Flow Publication Errors ................... 220
  7.5 Evaluation ................................................... 223
    7.5.1 STM Runtime Configuration ........................... 223
7.5.2 Phased Privatization ........................................ 224
7.5.3 Worklist Privatization ...................................... 225
7.5.4 Indirect Publication .......................................... 226
7.6 Summary .......................................................... 229

8 Conclusions and Future Directions 231

8.1 Contributions .................................................... 232
  8.1.1 Latency ....................................................... 232
  8.1.2 Throughput .................................................. 233
  8.1.3 Generality .................................................... 234

8.2 Future Directions ............................................... 235
  8.2.1 Thread-Level Speculation with STM .................... 236
  8.2.2 Custom STM Runtimes for Systems Software .......... 237
  8.2.3 Dynamic Selection of STM Runtimes .................... 238
  8.2.4 Lightweight Hardware Support for Transactional Memory . 239

Bibliography 241
List of Tables

4.1 A simple API for word-based STM. ........................................ 98
4.2 Prevalidation fence counts for STAMP benchmarks at different optimization levels. ........................................ 119
4.3 Postvalidation fence counts for STAMP benchmarks at different optimization levels. ........................................ 119
5.1 Interface between priority manager and STM library .................. 151
6.1 Summary of benefits and drawbacks of different inevitability options. 180
6.2 Overheads imposed by inevitability mechanisms. ....................... 180
List of Figures

2.1 RSTM metadata. .................................................. 41
2.2 LFUCache performance. .......................................... 52
2.3 RandomGraph performance. ..................................... 53
2.4 Sorted List Workloads ........................................... 54
2.5 Sorted List Workloads with Early Release ..................... 55
2.6 Red-Black Tree workloads ....................................... 57
2.7 Hash Table ....................................................... 58

3.1 Example ring holding 8 entries. ............................... 70
3.2 Red-black tree storing 20-bit integers; 50% of transactions are read-only. 88
3.3 RingSTM red-black tree commit/abort ratio. .................. 89
3.4 RandomGraph benchmark with 50% node inserts and 50% node re-
moval. .......................................................... 91
3.5 Hash table benchmark with 256 buckets, 8 bit keys, and 33% lookups. 91
4.1 Latency varies with the processor’s memory consistency model. .... 96
4.2 STAMP Genome benchmark. .................................... 120
4.3 STAMP Vaction (low contention) ...................................... 122
4.4 STAMP Intruder, RingSTM runtime. ................................. 123
4.5 STAMP KMeans (high contention), RingSTM runtime. ........ 124
4.6 STAMP Labyrinth, RingSTM runtime. ............................... 125
4.7 Varying frequency of postvalidation improves Labyrinth performance. 126

5.1 Workload with large write sets, for which lazy STM appears to perform badly. .................................................. 135
5.2 Linked list workload, which appears to favor eager acquire. .... 138
5.3 RandomGraph benchmark, for which the eager/passive strategy performs very well. ...................................... 142
5.4 Pathological doubly-linked list microbenchmark with 100% writer transactions. .................................................. 143
5.5 Throughput for a forest microbenchmark demonstrates the lack of “performance composability” for eager STM. ................. 144
5.6 The cost of priority-based starvation freedom in a pathological microbenchmark. ................................................ 158
5.7 Extensible hashing benchmark. ........................................... 161

6.1 State Transitions for the Writer Drain. ............................... 175
6.2 Inevitability Workloads .................................................... 190
6.3 Scalable worklist benchmarks. .......................................... 193

7.1 Privatization microbenchmarks. ....................................... 227
7.2 Indirect publication, modeled as extensible hashing with 256 tables. 228
List of Algorithms

1.1 Workload with two threads incrementing a shared counter . . . . . . . . 4
1.2 Implementation of synchronization operations with the CAS instruction. 6
1.3 Atomic increment of a variable using CAS . . . . . . . . . . . . . . . . . 6
1.4 A simple workload prone to deadlock . . . . . . . . . . . . . . . . . . . 9
1.5 Shared counter example, using transactions . . . . . . . . . . . . . . . 14
1.6 Abstract ownership record type . . . . . . . . . . . . . . . . . . . . . . 16
1.7 Translation of an atomic block to support rollback . . . . . . . . . . 16
1.8 Per-access instrumentation for a simple word-based STM algorithm . . 18
1.9 Boundary instrumentation for a simple word-based STM algorithm . . 19

3.1 RingSTM metadata . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
3.2 Read and write instrumentation, common to all RingSTM variants . . 78
3.3 RingSTM begin, end, and validation operations . . . . . . . . . . . . 79
3.4 Pseudocode for RingSW . . . . . . . . . . . . . . . . . . . . . . . . . 82
3.5 Pseudocode for RingR . . . . . . . . . . . . . . . . . . . . . . . . . . 85

4.1 Naïve and optimized transformation of two transactional reads, using a TL2-like runtime . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
4.2 Eliminating fences for reads issued within a loop . . . . . . . . . . . . 112
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Dynamically checked, un-fenced array indexing when array bounds are</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>known.</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Hoisting a transactional read. This transformation can cause incorrect</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>behavior.</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Add, Remove, and Print transactions for a simple set.</td>
<td>182</td>
</tr>
<tr>
<td>6.2</td>
<td>Parallel workloads for an array of sets.</td>
<td>183</td>
</tr>
<tr>
<td>7.1</td>
<td>Examples of publication and privatization.</td>
<td>203</td>
</tr>
<tr>
<td>7.2</td>
<td>Publication in parallel with safe and unsafe use of published data.</td>
<td>205</td>
</tr>
<tr>
<td>7.3</td>
<td>Publication by antidependence.</td>
<td>211</td>
</tr>
<tr>
<td>7.4</td>
<td>Unnecessary ordering in ALA.</td>
<td>212</td>
</tr>
<tr>
<td>7.5</td>
<td>Polling to detect doomed transactions.</td>
<td>216</td>
</tr>
<tr>
<td>7.6</td>
<td>An implementation of commit linearization in which read-only trans-</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>actions do not update shared metadata.</td>
<td></td>
</tr>
<tr>
<td>7.7</td>
<td>The commit fence.</td>
<td>219</td>
</tr>
<tr>
<td>7.8</td>
<td>Polling to detect publication.</td>
<td>221</td>
</tr>
<tr>
<td>7.9</td>
<td>TL2-style timestamps.</td>
<td>222</td>
</tr>
</tbody>
</table>
1 Introduction

Computer programs are typically written as a set of operations that a microprocessor executes in a fixed sequential order. This is the dominant way that programming is taught, and it is the most prevalent methodology practiced by hundreds of thousands, if not millions, of computer programmers throughout the world.

In the past, one could expect such a sequential program to run faster on a newer microprocessor. This performance improvement was due to two key characteristics. First, the clock rate of chips steadily increased for forty years. Since the inverse of the clock rate indicates (albeit coarsely) the amount of time to perform a single operation, a higher clock rate suggests that each operation takes less time. If we consider a running program to be a set of dynamically determined, sequentially ordered operations, then if each operation takes less time, the total execution will take less time.

Instruction-level parallelism (ILP) was the second factor providing improved performance. At a high level, ILP techniques within the microprocessor identified within any serial stream of instructions those operations that could be performed in parallel (at the same time) while maintaining the illusion of sequential execution. ILP enabled
chips to execute multiple instructions at the same time, effectively decreasing the length of the stream of instructions, and hence the time required to execute a program.

Both these sources of increased performance appear to have run dry. Higher clock rates consume too much energy and create more heat than can be safely dissipated; new ILP techniques require ever-increasing complexity for ever-diminishing returns. Since 2004, there has been neither an appreciable increase in clock rates of chips, nor an increase in the ability of commercial chips to find and exploit more ILP. In fact, clock rates have actually dropped, and some new processors are less aggressive than their predecessors at finding ILP. While modern microprocessors are being built with more transistors than ever before, these transistors are being used to provide multiple computing “cores” on a single chip. In many cases, the individual cores run at a lower frequency than their 2004 counterparts, thus consuming less power.

Unfortunately, a program written as a single stream of sequential operations can only use a single processing core, and thus will not run faster on a multicore chip than on a traditional single-core chip. The only programs that can expect to run faster on chips with high core counts are those that exhibit significant coarse-grained parallelism.

Often, a parallel program requires interaction among its component tasks. This interaction usually requires synchronization, which takes one of two forms: either one task must wait on a precondition satisfied by another task (“condition synchronization”) or a task must be sure that a set of updates it performs to shared memory occur without interruption by any concurrent tasks (“atomicity”) [Sco09, Chapter 12]. For forty years, atomicity has been achieved primarily though locks [Dij65]. Writing lock-based code has never been easy, and is typically left to a small set of highly skilled programmers. As parallel programming becomes ubiquitous, writing correctly synchronized code will become a core competency for all programmers, though most programmers lack the
skill to write lock-based code. Without an easy-to-use, high-performance mechanism for ensuring atomicity, programmers will not be able to use multiple cores, programs will cease to speed up when run on new processors, and our ability to solve new, large problems by expressing them as computations will decrease, ending the digital revolution that has dominated technological advancement for over half a century.

1.1 The Synchronization Problem

Breaking a program into tasks that can run in parallel is, itself, a large and unsolved problem. It is also, in many ways, dependent on which synchronization mechanisms are available to the programmer. If there is no general mechanism for sharing data among multiple tasks, then only programs that exhibit disjoint access parallelism (that is, no task performs a write to memory that is also read or written by a concurrently-executing task) can be parallelized. When atomicity is provided through mutual exclusion locks, only one task can access a particular memory region at a time. As long as concurrent access to a shared datum is unlikely, mutual exclusion locks can form the basis of highly scalable algorithms (as in many of the SPLASH-2 benchmarks [WOT+95]). Other mechanisms that provide atomicity (such as Transactional Memory, the focus of this dissertation), promise to allow larger classes of code to be parallelized.

To illustrate the subtlety of writing correctly synchronized code, Listing 1.1 depicts a program with two threads, each of which wishes to increment the same variable. Although the language-level instruction to increment variable $x$ is a single statement, the actual sequence of machine instructions entails first a load of variable $x$ into a register, then an addition, and finally a store of the result back into variable $x$. 
Listing 1.1: Workload with two threads incrementing a shared counter. The right hand side depicts the machine instructions that perform the increment of variable $x$.

When the program attempts to call this function from two concurrent execution contexts (“threads”), there are two possible outcomes. If one thread performs its store instruction (\texttt{st}) before the other performs its load (\texttt{ld}), then the result will be 2. However, suppose the two threads both perform the load before either performs a store. Both will observe $x = 0$, and both will compute the value 1. Regardless of the order in which the store instructions are performed by the two threads, the result will not be 2. From an external perspective, it appears that one thread never happened.

Programmer-centric memory consistency models [AG96] state that our example program is “incorrectly synchronized” due to a “data race”. Informally, a \textit{race} is the condition that correct program output is dependent on the interleaving of operations among multiple threads. A \textit{data race} is the condition that a pair of conflicting accesses to program data, performed by separate threads, are adjacent in a sequentially consistent [Lam79] history of the program’s execution [AH93]. In such a program, from one execution to the next, those two memory accesses could occur in either possible order \textit{even if all other memory accesses made by the program remained the same}. Data races typically indicate program bugs [MPA05], since they allow nondeterminism.
The solution to data races is to introduce synchronizing operations, that is, operations in each thread that guarantee an order between the threads. For example, suppose the load instruction is preceded by some sort of synchronization operation (an “acquire”), and the store is succeeded by another synchronization operation (a “release”). As long as the system guarantees that no acquire in thread $A$ may occur between an acquire/release pair in thread $B$, then so long as the operations within the acquire/release pair are not reordered with respect to the synchronization operations, the race condition (which occurs because $A$’s load occurs after $B$’s load but before $B$’s store) is avoided and the program is said to be “data race free”.$^1$

1.1.1 Hardware Instructions for Synchronization

The above presentation assumed that the acquire and release operations had global effect, but did not operate on actual program data or metadata. Modern architectures often provide low-level primitives that participate in a global synchronization order while also operating on program data. These “read-modify-write” (RMW) operations typically read an address, compute a new value, and store that value all in a single operation. The most prominent RMW operation is the compare-and-swap operation (CAS). Briefly, the CAS instruction reads an address, compares its contents to some expected value, and if the contents match, stores a new value, all in a single operation. In addition to performing these steps at once, CAS instructions are ordered with respect to all other operations on the same core, and also some operations on other cores. Typically, this ordering is sufficient to ensure correctness for CPUs that use the Total Store Ordering or Processor Consistency memory consistency models [AG96].

$^1$There exist more strict memory consistency models, such as those that require deterministic execution [DLCO09; MHKT09; OAA09]. Such models would not allow our program, since there is no guarantee as to which thread performs the first increment of $x$. 
On architectures such as the SPARC and Intel IA-32, \texttt{CAS} takes three operands: the address to be evaluated, the expected value of that address, and the replacement value. \texttt{CAS} returns the value stored in the address at the time when the \texttt{CAS} executed. A program can test this value against the expected value to determine whether the operation succeeded or not. Listing 1.2 demonstrates how \texttt{CAS} can be used along with some program-level metadata ($x'$) to implement the acquire and release operations.\textsuperscript{23}

For a simple operation such as atomic increment of a variable, more complex uses of \texttt{CAS} are possible, which perform the synchronization and data update at the same time. Listing 1.3 shows one possible implementation of a \texttt{fetch-and-increment()} operation, using \texttt{CAS}.

Throughout this dissertation, our presentation of algorithms will assume the availability of a \texttt{CAS} RMW operation. On architectures without \texttt{CAS}, such as POWER, the \texttt{load-linked/store-conditional (LL/SC)} pair of instructions can be used (along with a handful of additional instructions, to include a loop) to implement the

\textsuperscript{2}On many architectures there are simpler instructions that atomically modify a single bit.
\textsuperscript{3}On processors with relaxed memory consistency, the above example will require explicit memory fences to ensure ordering after the successful \texttt{CAS} and before the release operation.
same functionality as the \texttt{CAS} operation.\footnote{\texttt{CAS} and \texttt{LL/SC} provide different formal guarantees with respect to both spurious failures and unanticipated successes (the “ABA problem”). The algorithms in this dissertation are not affected by these differences.} Although the IA-32 provides a number of special-purpose RMW instructions (such as \texttt{lock inc} and \texttt{lock xadd}, which provide single-instruction increment and \texttt{fetch-and-increment()}, respectively), these special instructions can all be implemented with \texttt{CAS}, and so we will not consider these hardware instructions either.

\subsection{1.1.2 Locking and Nonblocking Synchronization}

The mechanisms presented in Listing 1.2 and Listing 1.3 both correctly protect our shared counter, but they do so in very different ways. In the first case, the programmer has created a locking protocol: some explicit metadata ($x'$) is associated with variable $x$, all accesses to $x$ must be preceded by an atomic update of $x'$ from 0 to 1, and all accesses to $x$ must be succeeded by a reset of $x'$ to 0 (note that a \texttt{CAS} is not required to perform the reset). So long as every access to $x$ obeys this protocol, use of $x$ by multiple threads will not result in a data race. Furthermore, this technique generalizes across operations and types for $x$: $x$ may be a single byte, a word, or a complex linked data structure, and the operation may be arbitrarily complex (indeed, the halting problem may reduce to it!). So long as all accesses to $x$ obey the locking protocol, any use of $x$ will not result in a data race.

The \texttt{fetch-and-increment()} operation does not provide such generality; it works only for performing an increment to an integer.\footnote{Note, however, that \texttt{CAS} can be used to construct very complex \texttt{fetch-and-phi()} operations.} However, it is \textit{nonblocking} (specifically lock-free [Her90]). At all times, all threads attempting to perform the \texttt{fetch-and-increment()} operation can be guaranteed that either they will succeed in a bounded number of processor clock cycles, or else they will fail because
some other thread succeeded in performing an increment during that duration of time. In contrast, our lock-based protocol provides no progress assurance. After successfully calling `acquire`, a thread could delay for an unbounded period of time, during which no thread can safely access \( x \), regardless of its type or the operation being performed by the lock holder.

When nonblocking techniques are known, they are generally preferred to their locking counterparts for a number of reasons. First, they provide strong progress guarantees (e.g., lock-free algorithms [Her90] guarantee that a thread attempting to perform an operation will either succeed, or fail because some other thread succeeded during the same time period). Second, they are immune to many pathologies experienced by locks (as discussed in Section 1.1.3). Third, a carefully designed nonblocking data structure typically has lower latency than its lock-based equivalent (e.g., lock-free queues [MS96] and lock-free allocators [Mic04]).

Unfortunately, neither lock-based code nor ad hoc nonblocking algorithms suffice for general purpose parallel programming. With ad hoc nonblocking synchronization, the technique for making a set of operations nonblocking for a single data structure is nontrivial. Universal constructions of nonblocking algorithms and data structures [Her90; Her93] tend to incur unacceptable latency [Mar08, Section 2.1]. With lock-based synchronization, either all uses of all shared variables must be protected by the same lock (an approach that may not scale), or else multiple locks must be used, introducing the risk of deadlock.

### 1.1.3 Problems With Locks

When as few as two locks are used by as few as two threads, there is the potential to fall into a situation where no thread can make progress. Listing 1.4 allows this situation,
struct locked_int
    lock l
    int i

locked_int a, b

transfer(locked_int* A, locked_int* B):
    1 acquire(A.lock)
    2 if (A.i > 0)
    3 A.i--
    4 acquire(B.lock)
    5 B.i++
    6 release(B.lock)
    7 release(A.lock)

main:
    1 ... // initialize the values of A.i and B.i
    2 t1 = createThread(transfer(&a, &b))
    3 t2 = createThread(transfer(&b, &a))

Listing 1.4: A simple workload prone to deadlock.

called deadlock, if both a.i and b.i are nonzero when the threads are created. It is possible for the first thread to lock a while the second is simultaneously locking b. Since both threads will then need to lock the other variable, both will wait until the corresponding lock is released. There is a cyclic dependency (neither thread can release the lock it holds until after it acquires the lock that it does not hold), and both threads will wait indefinitely. This problem can be avoided if there is a strict order in which locks are acquired (e.g., always acquire the lock for a before acquiring the lock for b). However, determining such an order is not easy, especially when the set of variables accessed by a region is not known statically.

Note, too, that a thread cannot release the first lock before acquiring the second. Were it to do so, it would be possible for some third thread to observe an incorrect program state. Suppose that a.i == b.i == 1 when the two threads call
transfer(). If the two threads each released the first lock after decrementing, a third thread could observe a state in which \( a.i + b.i = 0 \).

With only a single lock, deadlock is possible only if a thread attempts to acquire the same lock twice, and even this is easily prevented with reentrant locks.\(^6\) However, with only a single lock there will be limited scalability. If the program contained an array with thousands of `locked_ints`, then the likelihood of simultaneous access to the same variable by the two threads could be quite low. To be conservative, though, with a single lock any access to any `locked_int` would preclude access by any other thread to any other `locked_int`.

Additional pathologies may arise in lock-based synchronization, though they are less general and more easily remedied. In the convoying phenomenon, all threads acquire and release some set of locks in an approved order, but the existence of one lock holder \( L \) anywhere in the chain prevents any other thread \( T \) from reaching a lock further along in the chain until \( L \) completes; in effect, all threads are lined up and progressing in lock step, despite the potential for concurrent execution. With priority inversion, the lock holder \( L \) has lower priority than some thread \( T \) that requires the held lock to progress. Due to \( T \)’s higher priority, the runtime system does not allow \( L \) to run long enough to complete, and thus \( T \) waits even longer for \( L \) to complete.

### 1.1.4 Composability

Given two correctly synchronized data structure operations, a programmer may still not be able to create correct, race-free programs if there is no mechanism for composition. For example, in Listing 1.4, it did not matter that our `locked_int` type provided

\[^6\text{A reentrant lock stores the ID of the lock holder and a counter. Attempts by the lock holder to re-acquire the lock increment the counter, rather than blocking. Attempts to release the lock decrement the counter, and then release the lock only when the counter reaches zero.}\]
atomicity for any operation on a single instance. As soon as we needed to provide atom-
icity for an operation using two distinct locked int instances, our existing code was insufficient. We could not simply perform a locked decrement of A.i and then a locked increment of B.i: after the decrement, no synchronization is available to ensure that no other thread reads A.i before B.i is incremented. Nonblocking synchronization also fails for the same reason. In this case, programmers are often left with no option but to use a single lock for all synchronization, resulting in substantially lower scalability than is otherwise possible.

1.2 The Case for Atomicity

Our discussion of lock-based and nonblocking synchronization rapidly devolved into one of mechanism, rather than policy. In so doing, we also overlooked the ac-
tual properties provided by the competing mechanisms. Lock-based synchronization typically provides mutual exclusion, the condition that at most one thread is accessing the data protected by some lock at any time. Relaxations of mutual exclusion, such as Reader/Writer (R/W) locks, may provide the guarantee that the protected data are either being accessed in read-only mode by multiple threads, or in mutual exclusion mode by a single thread. In contrast, nonblocking operations provide a guarantee of linearizability [HW90]: a successful operation appears to happen at a single moment in time from the perspective of all other threads, and that moment is contained within the actual time between the invocation of the operation and its return.

These two mechanisms share a common characteristic: in both cases, the high-level operation appears to run atomically: in a properly synchronized program, no other thread is able to observe the intermediate state of a variable during any protected opera-
tion, and all threads agree on the order of that operation with respect to their operations
on the same data. If we cease to require that programmers indicate the mechanism
through which atomicity is provided, then questions of how to provide atomicity can
be left to a run-time system (or special hardware). The programmer need only specify
which code regions require atomicity. This offers many benefits.

• The programmer need not think about fine-grained locking protocols. Ideally,
  atomic sections are as simple to identify as coarse-grained locking protocols, but
  with scalability that rivals fine-grained locks.

• The run-time system can use information that is not statically available to choose
  the best mechanism for a given workload. This information may include specific
  hardware functionality, the likelihood of conflicts for a particular input data set,
  profile-based statistics describing the behavior of different phases of program ex-
  ecution, and the complexity of the atomic regions. The runtime can also evaluate
  atomic regions based on the resources they need, such as I/O devices.

• Problems of composability and deadlock disappear. Composition of multiple
  atomic regions can be provided simply by nesting those regions within a larger
  atomic region; the runtime must provide atomicity for the entire nested region.

• Priority inversion and convoying are easily avoided. A variety of mechanisms
  (some of them nonblocking) may be used by the runtime to implement atomicity.
  These mechanisms can be crafted to detect and prevent priority inversion and
  convoying, without any involvement by the programmer.
1.3 Atomicity Through Transactional Memory

Borrowing from the database community, Transactional Memory (TM) [HM93] introduces a language-level construct (atomic blocks) through which the programmer identifies regions of code that require atomicity to execute correctly. An underlying runtime system, which may leverage special hardware, then ensures that atomic blocks execute correctly, while exploiting parallelism whenever possible.

The promise of TM lies in its ease of use: programmers are freed from worrying about deadlock, mutual exclusion, composability, and data races. They need only annotate blocks of code whose accesses to shared memory ought to happen atomically; the language and runtime system then determine how to provide atomicity for those accesses in the face of concurrent execution of other atomic blocks.

A TM runtime be implemented entirely in software (software TM, or STM) [DS07; DSS06; Enn06; Fra03; FFR08; FH07; GHP05a; HF03; HLMS03; HMPH05; HPST06; MM08; MSH+06a; MSS05; SATH+06; ST95; WCW+07] or entirely in hardware (hardware TM, or HTM) [AAK+05; HWC+04; MBM+06; RHL05]. An STM may take advantage of specialized hardware to accelerate STM (hardware accelerated STM, or HASTM) [MTC+07; SATJ06; SDS08; SMD+06b; SSD+07; SSH+07]. Similarly, an HTM may only execute small, short transactions in hardware, and fall back to STM for transactions with large memory and time footprints (best-effort TM) [DFL+06; KCH+06; Moi05; MMN08]. In this dissertation, we focus exclusively on software TM: in addition to providing the only possible implementation of TM on millions of existing machines, software TM provides a framework to help identify what mechanisms truly belong in hardware, and thus avoids significant investment in complex hardware that ultimately proves unnecessary.
Listing 1.5: Shared counter example, using transactions. The access to variable x is placed within a lexically scoped atomic block. The programmer can expect the underlying runtime to ensure that concurrent accesses to x from within transactions will not result in data races.

Regardless of how the TM runtime is implemented, it must provide three properties from the database transaction model [GR93]. First, it must ensure that an atomic block appears to happen either in its entirety, or not at all (Atomicity). Second, the TM runtime must guarantee that (1) an in-progress instance of an atomic block (a “transaction”) never observes state changes caused by concurrent transactions, and (2) no in-progress transaction can see the intermediate state of another in-progress transaction (Isolation). Third, a transaction must move the system from one state in which all invariants hold to another state in which they also hold (Consistency).

1.3.1 Abstract STM Execution

In this dissertation, we will address the challenges of making STM both fast and general. Before doing so, we will describe a basic STM implementation, to ground the subsequent chapters in a consistent and real algorithm.

Listing 1.5 shows how our previous shared counter workload would be written with atomic blocks. While code within an atomic region is being executed, we will refer
to the dynamic instance as an *active* transaction. When the thread leaves the atomic region, we will refer to the transaction as *committed*. If two active transactions attempt to access the same data, and at least one active transaction is attempting to write that data, we will say that there is a *conflict*. In our example implementation, we will dynamically detect conflicts, and resolve conflicts by *aborting* one transaction. When a transaction aborts, the STM runtime will roll back the transaction’s local state and restart execution from the beginning of the atomic block.

### 1.3.2 Example Implementation

To illustrate the key challenges in implementing a fast, scalable transactional memory, we briefly describe an implementation that shares common features with most published STM algorithms that detect conflicts at the granularity of individual memory words [DS07; DSS06; FFR08; HPST06; SATH+06; WCW+07]. For simplicity, we consider a blocking algorithm, though nonblocking variants also exist [MM08; HF03].

The STM uses a global shared integer $C$ as a logical clock [DSS06; RFF06; RFF07]. The value of $C$ is initially zero, and increments by one whenever a transaction attempts to commit. As a shared memory counter accessed by all threads, $C$ is a scalability bottleneck, especially when all processing cores are not located on a single chip (as in symmetric multiprocessors), but we will ignore this potential source of latency for this discussion.

All conflicts are expressed and detected through accesses to a global table of *ownership records* (orecs), whose structure is defined in Listing 1.6. A simple hash function $H$ maps word-aligned addresses to entries within this table. An individual orec $O$ stores either the identity of a transaction attempting to modify a location covered by $O$, or else the logical time (as determined by the global counter $C$) at which a transaction com-
Listing 1.6: Abstract ownership record type. When the lock bit is set, the remaining 31 bits indicate the identity of the lock holder. Otherwise, the remaining bits indicate the time at which the orec was last acquired by a transaction that successfully committed.

```
struct orec
union
  unsigned time:31
  unsigned owner:31
  unsigned lock:1
```

Listing 1.7: Translation of an atomic block to support restart. On the left hand side, the programmer places code within an `atomic` block. On the right hand side, the boundaries of the `atomic` block are transformed so that exceptions thrown from within the STM library can unwind the stack and restart the transaction.

```
atomic while (true)
  ...
  // user code
  try
    transaction.begin()
    ...
    // user code (transformed)
    transaction.commit()
    break
  catch (Rollback)
    transaction.abort()
```

In our example algorithm, a transaction executes speculatively, buffering its stores in a private “write set” and logging the locations it reads in a “read set”. Our presentation uses exceptions for rollback, mapping a programmer specified transaction to a TM API according to the pattern shown in Listing 1.7.

Within the transaction, user code is transformed in three ways. First, stores to upward-exposed variables [CT03, page 437] that are thread-local (that is, those thread-local variables defined before the transaction body with values that may be overwritten by the transaction) are checkpointed to support rollback. When thread-local variable

MITTED CHANGES TO SOME LOCATION COVERED BY \( O \). Orecs provide all concurrency control, and ensure either a single writer or multiple readers access a location at any time.
$V$ is written by an in-flight transaction, this instrumentation typically stores the old value of $V$ to a log and performs the write in place. If the transaction commits, the log is discarded. Otherwise, the catch block of Listing 1.7 must use the undo log to return local variables to their pre-transaction-attempt state. This instrumentation is well understood, and we do not consider it further in this dissertation.

Second, in languages without garbage collection, all memory management instructions require instrumentation. The return value of every call to `malloc` made from within a transaction must be logged, so that the allocated regions can be deallocated on abort. Additionally, calls to `free` must be delayed until the transaction actually commits. Again, this instrumentation is well understood in the context of both nonblocking [Fra03, Section 5.2.3] and blocking [HSATH06] STM, and will not be considered further.

Lastly, individual loads and stores to addresses that are not provably thread-local, such as global and heap variables, require per-access instrumentation. For stores, the instrumentation serves to make changes to memory speculative by delaying the actual update of main memory until after the transaction logically commits. For loads, the instrumentation ensures that each load is consistent with all previous loads made by the active transaction. Additionally, load instrumentation must check the write buffer in the event of a load to a location already speculatively written by the transaction. This instrumentation appears in Listing 1.8.

In addition to per-access instrumentation, transactions require instrumentation at their boundaries (when they begin, commit, and abort). This instrumentation (which corresponds to the transformation in Listing 1.7) appears in Listing 1.9. At begin time, the transaction need only ensure its metadata lists are clear, and that it has an up-to-date copy of the global clock. The commit operation implements a two-phase locking
Listing 1.8: Per-access instrumentation for a simple word-based STM algorithm.

protocol [EGLT76] with optimistic consistency control [KR81]. To abort, the transaction releases any locks it might hold (due to rollback during the commit protocol). The complexity of the algorithm resides in two functions: transaction.read() and transaction.commit(), described in further detail below.
transaction.begin():
  1  readvector.clear()
  2  writehash.clear()
  3  lockvector.clear()
  4  start_time = global_clock

transaction.commit():
  1  for address in writehash
  2    orec_val ov = orec_table[hash(address)]
  3    if ov.lock
  4      if ov.owner != me
  5        throw Rollback
  6    else if ov.time > start_time
  7      throw Rollback
  8    else if !CAS(&orec_table[hash(address)], ov, <me, 1>)
  9      throw Rollback
 10   else
 11     lockvector.add(&orec_table[hash(address)], ov.time)
 12     write-readwrite-fence
 13   t = 1 + fetch-and-increment(global_clock)
 14    if (t != start_time + 1)
 15      for address in readvector
 16        orec_val ov = orec_table[hash(address)]
 17        if ov.lock
 18          if ov.owner != me
 19            throw Rollback
 20        else if ov.time > start_time
 21          throw Rollback
 22      for <address, value> in writehash
 23        *address = value
 24     write-write-fence
 25    for <address, val> in lockvector
 26      *address = <t, unlocked>

transaction.abort():
  1  for <address, val> in lockvector
  2    *address = <val, 0>
  3  ... // restore upward-exposed variables

Listing 1.9: Boundary instrumentation for a simple word-based STM algorithm.

The Read Operation  transaction.read() provides two guarantees: first, a read to a location previously written by the active transaction $A$ will always return the
value written by $A$, even though that value is stored in a speculative buffer. This property is provided by lines 1–2. The second guarantee is more complicated: the function ensures that all previous reads performed by $A$ are still valid: that is, no concurrent transaction committed changes to any location read by $A$ during $A$’s current execution attempt. This property is crucial for atomicity, since it ensures that all reads by $A$ should appear to occur at one logical time. It also provides opacity [GK08], a guarantee that the view of memory provided to an active transaction is consistent. The process of ensuring that all reads remain consistent is called validation. Note that in this algorithm, transactional reads must check metadata both before and after performing a dereference. Without both checks, a dereference cannot be certain to return the value written by the most recent committed transaction. Note that memory fences are required on processors with relaxed memory consistency, to ensure that the first metadata check precedes the read of program data on line 12, and to ensure that the second metadata check follows the read of program data.

**The Commit Operation**  Our implementation of `transaction.commit()` follows a simple protocol: after acquiring ownership of all locations in the write set, active transaction $A$ ensures that its reads remain valid. If so, $A$ is a logically committed transaction. It then replays all of its writes, transferring them from a private buffer into main memory, and then releases its locks. In this manner, the writes are isolated from other threads (that is, $A$’s writes are identifiable as speculative until $A$ is certain to have committed), and the acquisition of all locks before the release of any guarantees atomicity: all reads and writes appear to happen after all locks are acquired; during this time, all writes are logically unreachable by concurrent threads, and no individual write can be seen (since its location remains locked) until the final values of all writes have been issued to main memory. Again, processors with relaxed memory consistency re-
quire fences to ensure ordering after acquiring locks (line 12) and before releasing locks (line 24).

1.3.3 Implementation Alternatives

The above algorithm uses buffered update, commit-time locking, and timestamp-based validation, and detects conflicts at the granularity of individual words. These are deliberate design choices, and a number of alternatives exist.

Locating Metadata STM algorithms designed for object-oriented languages typically do not use a table of ownership records [MSH+06a; HLMS03; MSS05; Fra03; FH07; GHP05a; RdB08; DZ09; HPST06]. Instead each language-level object has an ownership record embedded within it. In the Bartok STM [HPST06], the ownership record is superimposed over the existing language-level lock word. In Riegel’s work [RdB08], the ore is appended to the end of the object. By locating orecs alongside of data, there is less danger of false conflicts due to the choice of hash function or an insufficient allocation of orecs for an application’s memory requirements. There is also the potential for increased spatial locality, since the object and its metadata may reside in the same cache line. However, there is also the potential for false conflicts when two fields of an object (such as two entries within an array field) can safely be modified in parallel.

Acquisition Time Several STM algorithms opt to acquire the ore covering an address when that address is first passed to a transaction.write() call. With this “eager” acquisition of ownership, the cost of write-set lookup can be decreased: if a location’s ore is not locked, the lookup on lines 1–2 of Listing 1.8 is not needed, since the calling transaction could not possibly have a speculative store to that location.
**Speculative Write Storage**  With eager acquisition, a transaction may opt not to buffer its speculative writes. Instead, the transaction can maintain an undo log, holding the values that must be restored in the event that the transaction aborts, and then the transaction can perform its speculative writes in-place, much as it does writes to thread-local variables. The undo log is discarded upon transaction commit. If log entries are not filtered (to detect multiple transactional writes to the same field), then on abort, the log must be replayed in reverse to restore memory correctly.

**Consistency Mechanisms**  In our example algorithm, transactions used logical time to determine whether each read was consistent with all previous reads. This is not the only alternative. Early STM algorithms used quadratic validation, which checked all reads whenever any new location was read. Other STM algorithms use visible reads, a protocol in which transactions mark the locations that they read; a concurrent writer must explicitly abort active readers before acquiring a location, and readers must test if they have been aborted after every read. These techniques all ensure that a transaction is consistent at all times (a prerequisite for opacity), via incremental growth: on each read, the transaction determines whether the addition of that read to the set of previous reads would result in a larger set of reads that could still all have happened at the same logical time. Some STM algorithms do not provide this guarantee; instead they allow an in-flight transaction to observe and use an inconsistent view of memory, and rely upon run-time sandboxing to ensure that inconsistencies do not lead to externally visible effects, such as infinite loops, segmentation faults, or execution of unsafe code via function pointers.
1.4 Alternatives to Transactional Memory

Two of the most promising proposals for making parallel programs faster or easier to write are speculative lock elision and speculative parallelization of sequential code. Both have considerable overlap with TM. We outline the key differences below.

1.4.1 Speculative Lock Elision

Like TM, speculative lock elision (SLE) was first proposed as a hardware technique [MT02; RG01], with software proposals following [OCS07; RHH09]. Indeed, much recent hardware TM research stems directly from the SLE proposal of Rajwar and Goodman and the work of Martinez et al..

SLE increases the scalability of an existing, lock-based, parallel program by allowing critical sections to run in parallel, even when they are protected by the same lock. To achieve this goal, SLE uses a runtime system or custom hardware to buffer stores and to monitor the reads and writes made by the critical section, so that conflicts can be detected and resolved.

There are three differences between SLE and TM. First, SLE does not improve programmability. SLE takes as input a lock-based program, and attempts to increase scalability by speculatively running critical sections in parallel. Thus the programmer must think in terms of locks, and must ensure that an execution of the program is correct and performs well even on platforms without SLE support.

The second difference deals with disjoint access parallelism. For lock-based programs with multiple locks, SLE may offer greater performance than TM. Both TM and SLE may identify false conflicts among critical sections; however, in TM, all false conflicts must be treated as true conflicts. In a correctly synchronized program, if SLE
identifies a conflict between two regions, but those two regions are protected by different locks, then the conflict can be ignored in some cases [RHH09].

The third difference deals with semantics. Even in an application where all critical sections are protected by a single, global mutual exclusion lock, SLE and TM are not identical, since SLE demands lock-based semantics. Menon et al. show that the semantics of locks are far more restrictive than the semantics of most STM implementations, leading to significant run-time overheads [MBS+08a]. In Chapter 7 we explore the impact of relaxed semantics on performance. Since TM can provide relaxed semantics, while SLE must guarantee the same semantics as a lock-based execution of the input program, TM may offer superior performance.

### 1.4.2 Speculative Parallelization of Sequential Code

Whereas SLE takes as input a lock-based parallel program, systems that employ speculative parallelization attempt to automatically parallelize a sequential program. To ensure correctness, the parallelization must obey the sequential specification of the program. For applications with irregular access patterns and possible conflicts, the two leading approaches are hint-based parallelization [DSK+07] and implicit parallelization with ordered transactions [vPCC07].

In the Behavior Oriented Parallelism (BOP) system [DSK+07], the programmer annotates regions of code that may be able to run in parallel. These possibly parallel regions (PPRs) are then executed in a sandboxed environment, which leverages OS processes and page-level protection to avoid instrumentation of individual instructions. Annotations are treated as hints, and may be ignored by the runtime system. Furthermore, annotations are not required to be lexically nested. Additionally, since there are no per-access overheads or limitations, PPRs can be very large.
Futures provide another approach for hint-based parallelism. Halstead’s original proposal added futures to Scheme, primarily as a means of parallelizing expression evaluation in a setting where side effects were rare [Hal85]. Welc et al. subsequently added “safe” futures to Java [WJH05], using TM-like mechanisms to detect conflicts and guarantee that a parallel execution remained equivalent to that of the original serial program. In both cases, futures appear best suited to small regions of code that modify shared data infrequently.

The system of von Praun et al. [vPCC07] uses implicit hardware transactions to parallelize loop bodies. An underlying hardware TM provides sandboxing, and there is a rigid structure to the parallelization, which facilitates the use of well-known reductions. In order to guarantee sequential semantics, loop iterations commit in order. Unlike BOP, there is no mechanism for forwarding data between speculative threads.

While these mechanisms facilitate the parallelization of existing code, all suffer from a need for significant sandboxing. This sandboxing ensures that a speculative region does not cause an externally visible artifact when it attempts to use a datum that has not yet been initialized by a previous region that has not yet completed. Furthermore, no technique completely relieves the programmer of the burden of modifying an existing program to eliminate artificial bottlenecks. While this task is certainly simpler than rewriting an application to use transactions, it is tedious and often fails to identify profitable parallelizations.

### 1.4.3 Summary

Both SLE and speculative parallelism offer alternatives to TM for assisting in the creation of scalable programs. Each represents a different programmability/performance tradeoff: SLE focuses entirely on performance and does not address programmability
at all; TM eliminates reasoning about deadlock, but still requires the programmer to write multithreaded code and identify regions that require atomicity; speculative parallelization requires a minimal amount of programmer intervention, but often is limited in its ability to parallelize a program.

There are substantial similarities among the three approaches. In all cases, the resulting parallel program leverages speculation to resolve conflicts, and requires a runtime system or custom hardware to track or otherwise instrument all reads and writes. In Chapter 8 we suggest opportunities to exploit this overlap, so that the mechanisms presented in this dissertation can aid in the parallelization of existing programs.

1.5 Problems With STM Implementations

There are many STM runtimes built from the above framework. However, these systems have not yet been used to create large, highly-scalable applications. We briefly consider the problems that have prevented widespread adoption below.

1.5.1 Single-Thread Latency

In our candidate algorithm, each load or store requires expensive instrumentation. Each memory access requires at least a dozen instructions, and on processors with relaxed memory consistency, each read requires two memory fences, where each fence has both high latency and potential negative effect on instruction-level parallelism. Additionally, each store requires a CAS operation; typically CAS is expensive since it enforces memory ordering. Furthermore, if the instruction count for read and write instrumentation is too high, then the instrumentation may not be inlined, resulting in function call overhead on every memory access.
In addition to these per-access costs, our instrumentation risks introducing substantial overhead at transaction boundaries. Clearly the commit protocol is expensive, since it can introduce overhead linear in the number of reads (to validate or to release read visibility) and linear in the number of writes (to release write ownership). Additionally, there is a space cost to store large read/write sets, and the time overhead of maintaining those sets. For large transactions, validating $R$ reads may result in $R$ cache misses.

Latency can be reduced in three ways: first, by designing new algorithms that avoid some of these costs; second, by creating compiler optimizations to eliminate redundant instrumentation; and third, through hardware-software co-design, aimed at creating STM algorithms whose most expensive operations can be performed efficiently with special-purpose hardware. This dissertation will consider the first two approaches, and will suggest directions for the third.

1.5.2 Throughput and Generality

Our algorithm relies upon a shared global counter. This counter is a scalability bottleneck, since all transactions must access it when committing writes. Such bottlenecks can potentially be addressed with custom hardware. A more uncertain question deals with the scalability of transactions due to workload characteristics. In particular, when a set of transactions conflict, at least one must roll back in order for others to progress. Peak throughput requires that the aborted transaction (or transactions) be selected with great care, so that there are no unnecessary aborts. Ideally, the TM runtime will make this decision automatically, in a manner that maximizes throughput and fairness. In the worst case, the programmer may also need to divide an individual atomic block into several atomic blocks to reduce the frequency of conflicts among in-flight
transactions. This “performance debugging” is hopefully simpler than the correctness
debugging typically required of fine-grained locking protocols.

In Listing 1.8, we presented an implementation of atomic blocks that used spec-
ulation. Such an implementation cannot safely perform I/O, system calls, and other
irrevocable operations since there is no general means of undoing I/O or resetting op-
erating system state. In order for TM implementations to provide the full generality
of atomicity, they must support an inevitable mode (i.e., nonspeculative execution of
atomic blocks), in which some transactions are guaranteed to never abort. I/O and sys-
tem calls are safe from such transactions; however, the state of the art forbids other
transactions from executing while an inevitable transaction is active.

Furthermore, a transaction has no means of communicating with other active trans-
actions and allowing that communication to influence its behavior. The most widely-
accepted solution is to use explicit self-abort (via the 
retry command) as a means of
waiting for a condition to be met [HMPH05]. The aborting transaction can then mark
the locations it has read, so that it can yield the processor and be woken up after a
subsequent transaction modifies a marked location. 
retry is typically implemented
with visible readers, which require significant time and space overheads. Alternatives
to retry-based condition synchronization, such as Transaction Synchronizers [LM05],
transactional condition variables [DS09c], and punctuated transactions [SKBY07], have
seen only limited attention.

1.5.3 Interaction with Nontransactional Code

Our discussion up to this point ignored the potential for races between transactions
and nontransactional code. In our previous discussion of isolation, we stated that the
TM runtime must guarantee that in-progress transactions only observe state updates
performed by successful transactions. However, this definition ignores the possibility that nontransactional code might access the same state as transactional code. Often, TM implementations are divided into two categories: *Weakly Isolated* implementations guarantee only that transactions are isolated from each other; *Strongly Isolated* implementations also guarantee that transactions are isolated from state changes performed by nontransactional code, and vice-versa [BLM05; LR07]. Most HTM implementations provide strong atomicity, as do a handful of STM implementations [AHM09; SMAT+07; SMSAT08].

Strongly isolated TM implementations provide guarantees about the behavior of incorrectly synchronized applications (e.g., those that simultaneously access the same datum both transactionally and nontransactionally), where transactional accesses include both (1) reads and writes performed by an in-flight transaction and (2) reads and writes performed by the TM runtime during validation and commit operations. For correctly synchronized programs, the semantics of weakly isolated STM have often been described using single global lock atomicity [LR07], where “a program executes as if all transactions were protected by a single, program-wide mutual exclusion lock.” However, most weakly isolated STM implementations provide this guarantee only among transactions, and may introduce races between nontransactional operations and operations performed by the STM runtime itself [DS09a; MBS+08a; MSS08; SDMS08; SMDS07; WCW+07].

**The Privatization Problem** Let us suppose that transaction \( U \) updates several nodes in the suffix of a linked list and commits. Subsequently, transaction \( S \) severs a larger suffix of the list and commits. After committing \( S \), the program is guaranteed that there are no references to the suffix in other threads, and thus subsequent operations

---

\(^7\)Some authors prefer the terms “Strong Atomicity” and “Weak Atomicity”.
on the suffix do not require transactional instrumentation. However, if $U$ is still in
its commit protocol (after its logical commit point, but still performing write-back),
then its write-back can race with subsequent nontransactional accesses in the thread
that ran $S$ [LR07; SMDS07]. Another problem can occur if $U$ is still active when
$S$ commits. $S$’s commit will ultimately force $U$ to abort. If $U$ continues to run (as
a “doomed transaction”) while nontransactional updates are made to $S$, then $U$ might
perform externally visible operations due to inconsistent reads [SMDS07; WCW+07].

**Publication**  A similar problem can occur if a private location is modified nontrans-
actionally, and then it is transitioned to a shared state. If a concurrent transaction reads
the nontransactional data before the transition operation occurs (due to compiler re-
orderings or value speculation), then it may perform an invalid computation due to its
inability to detect conflicts with the nontransactional initialization code [MBS+08a; MBS+08b].

Mechanisms to prevent these problems implicitly, by assuming that all transactions
may publish or privatize data, can severely limit STM performance: such mechanisms
typically force all transactions to participate in a total global order, and also require
transactions to wait at their begin and end points to ensure that the global transaction
order is compatible with the implicit order between transactional and nontransactional
accesses within each thread. However, some STM algorithms are implicitly privatiza-
tion safe and/or publication safe.

---

8 An analogous problem exists for aborted transactions that have not completed their undo operation
in STM implementations that use in-place update.
1.6 Contributions

This dissertation presents novel techniques to improve STM in three areas: (i) decreasing single-thread latency, (ii) ensuring throughput and fairness, and (iii) increasing the generality of TM by supporting programs that perform I/O and syscalls within transactions, decreasing the overhead of retry-based condition synchronization, and facilitating low-overhead support for privatization and publication.

1.6.1 Decreasing Latency

The first practical STM implementations [Enn06; Fra03; FH07; GHP05a; HF03; HLMS03; HMPH05; MSH+06a; MSS05; SATH+06] used visible reads, sandboxing, or quadratic validation to ensure consistency. Subsequently, timestamps were adopted [DSS06; RFF07; WCW+07]; in all cases, the underlying STM used ownership records, and thus incurred high-constant overhead linear in both the number of reads and the number of writes. Furthermore, many of the metadata operations performed by these STM algorithms require expensive memory ordering guarantees.

We address these sources of latency directly. First, we propose a global commit counter mechanism [SMSS06]. Like visible reads and quadratic validation, the commit counter provides strong safety guarantees (both opacity and protection from half of the privatization problem). However, like timestamps and sandboxing, it incurs low overhead in the common case. We show how the commit counter can safely relax conflict detection mechanisms, thereby avoiding some resolvable conflicts, and we discuss the generality of our mechanism to object-based and word-based blocking and nonblocking STM.
We then introduce RingSTM [SMvP08], a livelock-free, privatization-safe, lock-based STM. RingSTM uses Bloom filters [Blo70] for conflict detection, avoiding the need to maintain large sets of ownership records. It also avoids the need for multiple CAS operations, and is amenable to hardware acceleration in a number of directions.

We present compiler optimizations that accelerate both RingSTM and traditional orecc-based STM [SMSW09]. Our mechanisms focus on eliminating redundant memory fences when running STM on processors with relaxed memory consistency. However, our techniques also decrease the number of instructions required to perform validation in RingSTM, and in orecc-based STM implementations that use a commit counter, quadratic validation, or visible reads.

### 1.6.2 Increasing Throughput

Eager STM typically uses an out-of-band contention manager to make conflict resolution decisions [GHP05a; HLMS03; SS05a]. These managers tend to require significant bookkeeping (introducing latency), but remain brittle to workload characteristics and rarely provide stronger guarantees than livelock avoidance.

We present a design strategy for STM that avoids livelock without sacrificing latency or requiring bookkeeping [SDMS09]. Our strategy is based on the use of lazy acquisition of ownership, but differs from previous work in that it does not introduce latency: by using extendable timestamps [RFF07] and an indexed write-set, our baseline lazy system has latency comparable to an in-place, direct update system. Given these characteristics, we use a simple conflict detection policy that defers to a committing lock-holder, but does so without introducing many unnecessary aborts.

We then turn to the greater challenge, of increasing fairness, and show how an user-defined priority and starvation avoidance can be implemented in an out-of-band
mechanism. Our technique can either use Bloom filters to track conflicts, or it can employ the visible reader bitmaps we originally developed for the nonblocking RSTM system [MSH+06a]. Furthermore, both mechanisms can also be used to implement low-latency retry-based condition synchronization [SSS08].

1.6.3 Increasing Generality Without Sacrificing Performance

When transaction \( T \) must perform I/O, make system calls, or interact with nontransactional code, the accepted solution is for \( T \) to first set a flag preventing new transactions from starting, and then wait for all active transactions to commit or abort. At this point \( T \) runs in isolation, and cannot abort; it is said to be inevitable, and it may do anything a lock-based critical section could do.

We demonstrate that transactions can run inevitably without forbidding concurrency [SMS08; SSD+08]. We present a suite of mechanisms that allow concurrency between an inevitable transaction and concurrent noninevitable transactions; the mechanisms differ as to what types of concurrent noninevitable transactions are allowed (read-only or read-write) and how those transactions respond to the appearance of a new inevitable transaction (blocking, aborting, completing).

The second challenge in making transactions more general regards the interaction between transactional and nontransactional code that accesses the same data. Such access is logically correct when transactions are used to transition data from a shared (transactional) state to a private state (and back again). However, few STM implementations support this behavior, and those that do require significant ordering delays on all transaction boundaries.

We propose a semantics for transactional programming languages that does not equate transactions with lock-based critical sections, but instead defines a total serial
order on all transactions [SDMS08]. We then show how programmer annotation on only those transactions that transition data between shared and private states can allow the runtime to ensure correctness without substantial overhead. In effect, our annotations elevate serializability to strict serializability, but only when it is required.

1.7 Roadmap

Chapter 2 establishes safety criteria for STM and presents the global commit counter mechanism, which decreases validation overhead through a lightweight polling operation. In Chapter 3 we present RingSTM, an algorithm that obeys these safety criteria but does not use ownership records. Chapter 4 then discusses compiler optimizations for ownership record-based STM and RingSTM; taken together, these three contributions decrease latency of individual transactions without decreasing safety. Furthermore, they do so in a way that improves the ability of specialized hardware to further accelerate STM.

In Chapter 5, we consider throughput, and discuss how an STM can be engineered to provide good progress guarantees without introducing latency. We then develop mechanisms to support priority and avoid starvation at low cost, and show how these mechanisms can be unified with inevitability and efficient retry. Chapter 6 considers the special case of using inevitable transactions to ensure throughput and provide support for nontransactional operations (such as I/O) within a transaction. Chapter 7 turns to the problem of language-level semantics, and discusses techniques to support publication and privatization without significant overhead. Chapter 8 concludes and discusses future research opportunities.
2 Safely Reducing The Cost of Conflict Detection

In an STM system, conflict detection is the problem of determining when two transactions cannot both safely commit. Validation is the main mechanism through which active transactions detect conflicts. At the very least, validation must ensure that conflicts are detected before a transaction commits, so that the transaction can abort before making incorrect updates to shared state. However, it is often necessary to perform validation more frequently: doing so can prevent an active transaction from viewing inconsistent data, which might potentially cause a destined-to-abort transaction to exhibit irreversible, externally visible side effects.\(^1\) Existing mechanisms for conflict detection vary greatly in their degree of speculation and their relative treatment of read-write and write-write conflicts. Validation, for its part, appears to be a dominant factor—perhaps the dominant factor—in the cost of complex transactions. We present a new mechanism, the global commit counter heuristic. Our heuristic can greatly reduce the cost of validation and single-threaded execution, while also providing strong safety guarantees. It also improves an STM’s ability to resolve conflicts profitably through mixed invalidation.

\(^1\)This protection affects both privatization-safety and opacity.
2.1 Techniques For Detecting and Resolving Conflicts

In recent work we introduced RSTM [MSH+06a], a lightweight, indirection-based, obstruction-free STM system. We also analyzed its costs [MSH+06b]. In addition to copying overhead, which appears to be unavoidable in nonblocking STM implementations that detect conflicts at the granularity of language-level objects [SSD+07; TWGM07], we found the two principal sources of overhead to be bookkeeping and incremental validation. Bookkeeping serves largely to implement conflict detection—that is, to identify pairs of concurrent transactions which, if permitted to commit, would not be linearizable [HW90]. Validation serves to ensure that transactions never see or make decisions based on inconsistent data; we use the term “incremental” to indicate strategies in which the overhead of each validation is proportional to the number of objects previously accessed.

Two concurrent transactions are said to conflict if they access the same object and at least one of them modifies that object. When an STM system identifies a conflict, it typically allows one transaction to continue, and delays or aborts the other. The choice may be based on a built-in policy (as, for example, in the lock-free OSTM [Fra03; FH07]), or it may be deferred to a separate contention manager (as, for example, in the obstruction-free DSTM [HLMS03]). Contention managers typically employ heuristics to achieve good throughput, especially for systems that use eager acquisition [GHKP05; GHP05a; GHP05b; SS04; SS05a; SS05b]. We will consider contention management in more detail in Chapter 5. In this chapter, we thoroughly and systematically study conflict detection and validation.

Conflict detection. An STM system may notice potential conflicts early in the life of the conflicting transactions, or it may delay such notice until one of the transactions
attempts to commit. The choice may depend on whether the conflict is between two writers or between a reader and a writer. In the latter case, it may further depend on whether the reader or the writer accesses the object first. If transactions $S$ and $T$ conflict, aborting $S$ early may avoid fruitless further computation. In general, however, there is no way to tell whether $T$ will ever commit; if it ultimately aborts, then $S$ might have been able to commit if it had been permitted to continue.

In a recent study, Scott analyzed the semantics of alternative strategies for conflict detection, and identified representative systems for each strategy [Sco06]. Briefly, the three dominant strategies are:

**visible-eager:** When a transaction $T$ wishes to read or write a location, it explicitly marks its intention in the location’s metadata. During execution, $T$ must perform contention management if it wishes to read a location marked for writing by another transaction, or if it wishes to write a location that is marked for reading or writing by another transaction.

**invisible-eager:** When reading, $T$ does not modify metadata. Instead, it periodically checks to ensure that no location it previously read has subsequently been marked for writing by a concurrent transaction. Contention management is required only when attempting to read a location marked for writing. A final check is required at commit time, to ensure that after all writes have been marked, all reads remain valid.

**invisible-lazy:** During execution, $T$ does not modify any metadata. At commit time, it marks all the locations it intends to write, and then checks that all its reads remain valid. Contention management is available, though typically only required to ensure nonblocking progress in the face of preemption.
(Note that visible-lazy is also possible, though typically not implemented).

Scott suggested that it might make sense to detect and resolve write-write conflicts early (since at most one of the conflicting transactions can ever commit), but read-write conflicts late (since both may commit if the reader does so first), via a hybrid strategy called mixed invalidation; to the best of our knowledge, it has not been explored in any prior TM system.

**Validation.** Since a transaction that commits successfully has no visible side effects prior to the commit, it is tempting to assume that an aborted transaction will have no visible effects whatsoever. Problems arise, however, in the presence of transaction conflicts. Suppose, for example, that $f()$ is a virtual method of class $A$, from which are derived subclasses $B$ and $C$. Suppose further that while $B.f()$ can safely be called in transactional code, $C.f()$ cannot (perhaps it performs I/O, acquires a lock, or modifies global data under the assumption that some lock is already held). Now suppose that transaction $T$ reads objects $x$ and $y$. Object $y$ contains a reference to an object of class $A$. Object $x$ contains information implying that the reference in $y$ points to a transaction-safe $B$ object. Unfortunately, after $T$ reads $x$ but before it reads $y$, another transaction modifies both objects, putting a $C$ reference into $y$ and recording this fact in $x$. Because $x$ has been modified, $T$ is doomed to abort. If it does not notice this fact right away, however, $T$ may read the $C$ reference in $y$ and call its unsafe method $f()$.

While this example is admittedly contrived,\(^2\) it illustrates a fundamental problem: even in a type-safe, managed language, a transaction that is about to perform a potentially unsafe operation must verify the continued validity of any previously read objects on which that operation has a control or data dependence.

---

\(^2\)Guerraoui et al. present additional examples inspired by this observation [GK08], such as doomed transactions entering infinite loops, generating divide-by-zero exceptions, or causing program termination due to segmentation faults and bus errors.
Unfortunately, straightforward *incremental validation*—checking all previously read objects on each new object reference—leads to \( O(n^2) \) total cost when opening \( n \) objects, an extraordinary burden for transactions that access many objects. Similarly, visible readers—which allow a writer to identify and explicitly abort the transactions with which it conflicts—incur very heavy metadata update costs (typically 2 **CAS** instructions per read) and cache eviction penalties [MSH+06a; SATH+06], or else require substantial space overhead [DS09b; LLM+09]; in our experiments with simple data structures, update and eviction costs appear worse than the quadratic cost of incremental validation. A third option, timestamp-based STM [DSS06; RFF06], maintains a logical clock and ensures that every successful transaction is assigned a unique commit time from that clock. By embedding the most recent commit time in each object, a transaction can ensure consistency without incurring \( O(n) \) overhead on each object access in the common case. However, straightforward uses of timestamps complicate the problem of providing privatization safety [SMDS07].

Static analysis of data flow, control flow, and type safety may allow a transaction-aware compiler to avoid some validation calls made by library-based STM [DMSS07]. However, the underlying STM is typically given only three validation options: (1) require the programmer to validate manually wherever necessary, (2) accept the quadratic cost of incremental validation, or (3) use timestamps. Option (1), we believe, is unacceptable: identifying the places that require validation is too much to expect of the typical programmer. We prefer instead to find ways to avoid or reduce the cost of incremental validation, preferably without complicating privatization safety.

**Contributions.** This chapter makes two principal contributions. First, we evaluate strategies for conflict detection in the context of a single STM system. We consider

---

3For some workloads, these problems can be mitigated with semi-visible reads [MSS08].
lazy acquire, in which conflicts are noticed only at commit time; eager acquire, in which conflicts are noticed as soon as two transactions attempt to use an object in incompatible ways; and mixed invalidation, in which conflicts are noticed early, but not acted upon until commit time in the read-write case. We also consider both visible and invisible readers. Invisible readers require less bookkeeping and induce fewer cache misses, but require that read-write conflicts be noticed by the reader. Visible readers allow such conflicts to be noticed by writers as well.

Second, we introduce a lightweight heuristic—the global commit counter—that eliminates much of the overhead of incremental validation. Specifically, transaction $T$ validates incrementally only if some other transaction has attempted to commit writes since $T$’s previous validation. In multithreaded experiments, the savings ranges from negligible in very short transactions to enormous in long-running applications (95% reduction in validation overhead for our RandomGraph “torture test”). Because it allows us to overlook the fact that a previously read object is being written by an as-yet-uncommitted transaction, the commit counter provides a natural approximation of mixed invalidation. It also allows us to notice when a transaction is running in isolation, and to safely elide bookkeeping of reads, validation, and contention management calls. This elision dramatically reduces the cost of STM in the single-threaded case.

Section 2.2 provides an overview of our RSTM system, including a description of eager and lazy acquire, visible and invisible readers, and mixed invalidation. Section 2.3 then presents the global commit counter heuristic. Performance results appear in Section 2.4, related work in Section 2.5, and conclusions in Section 2.6.
2.2 Overview of RSTM

The Rochester Software Transactional Memory System (RSTM) is a library-based framework for using STM in C++ programs. It seeks to provide a simple programming interface and facilitate experimentation. Experiments in this chapter use the non-blocking, object-based RSTM algorithm (other chapters use a variety of word-based algorithms, but the same API and benchmarks). Its metadata organization (Figure 2.1) roughly resembles that of DSTM [HLMS03], but with what the latter calls a “Locator” merged into the newest copy of the data. Detailed description can be found in a previous paper [MSH06a]; we survey the highlights here.

As in most other nonblocking object-based STM algorithms, an object is accessed through an object header, which allows transactions to identify the last committed version of the object and, when appropriate, the current speculative version. The metadata layout is optimized for read-heavy workloads; in the common case, the header points
directly to the current (shared, read-only) version of the object. When an object is being written, one additional level of indirection is needed to reach the last committed version.

Each thread maintains a transaction descriptor that indicates the status (active, committed, or aborted) of the thread’s most recent transaction, together with lists of objects opened (accessed) for reading and for writing. To minimize memory management overhead, descriptors are allocated statically and reused in the thread’s next transaction. RSTM currently supports nested transactions only via subsumption in the parent.

Data object versions are dynamically allocated from a special per-thread heap with lazy generational reclamation. As in OSTM [Fra03; FH07] or McRT [HSATH06; SATH+06], “deleted” objects are not reclaimed until every thread is known to have been outside any potentially conflicting transaction, or to have validated after the commit of the transaction performing the delete.

**Acquisition** A transaction never modifies a data object directly; instead, it clones the object and makes changes to the copy. At some point between open time (initial access) and commit time, the transaction must acquire the object by making the object header point to the new version of the data (which in turn points to the old). Since each new version points to the transaction’s descriptor, atomically CAS-ing the descriptor’s status from active to committed has the effect of updating every written object to its new version simultaneously. Eager (open-time) acquire allows conflicts to be detected early. As noted in Section 2.1, the timing of conflict detection enables a tradeoff between, on the one hand, avoiding fruitless work, and, on the other, avoiding spurious aborts.
Reader visibility  The programmer can specify whether reads should be visible or invisible. If reads are visible, the transaction requests one of 32 visible reader tokens. Then, when it opens an object for reading, the transaction sets the corresponding bit in the object’s visible reader bitmap. Thus while the system as a whole may contain an arbitrary number of threads, at most 32 of them can be visible readers concurrently (the rest can read invisibly). The bitmap is simpler and slightly faster than an alternative mechanism we have described [MSH+06a] that supports an arbitrary number of visible readers.

Before it can acquire an object for writing, a transaction must obtain permission from its contention manager to abort all visible readers. It performs these aborts immediately after acquisition. A transaction that has performed only visible reads is thus guaranteed that if it has not been aborted, all of its previously read objects are still valid. By contrast, as described in Section 2.1, an invisible reader must (absent static analysis) incrementally validate those objects on every subsequent open operation, at \(O(n^2)\) aggregate cost.

In practice, visible readers tend to cause a significant increase in memory traffic, since the write by which a reader announces its presence necessarily evicts the object header from every other reader’s cache. In several of our microbenchmarks, visible readers perform worse than invisible readers at all thread counts higher than one.

Mixed invalidation  If two transactions attempt to write the same object, one argument for allowing both to proceed (as in lazy acquire) holds that any execution history in which both remain active can, in principle, be extended such that either commits (aborting the other); there is no a priori way for an implementation to tell which transaction “ought” to fail. This is a weak argument, however, since both cannot succeed. When a reader and a writer conflict, however, there is a stronger argument for allowing
them to proceed concurrently: both can succeed if the reader commits first. We therefore consider a mixed invalidation strategy [Sco06] in which write-write conflicts are detected eagerly but read-write conflicts are ignored until commit time. The following section considers the implementation of mixed invalidation and a heuristic that cheaply approximates its behavior.

2.3 The Global Commit Counter Heuristic

As noted in Section 2.1, a transaction must validate its previously-opened objects whenever it is about to perform an operation that may be unsafe if the values of those objects are mutually inconsistent. We take the position that validation must be performed automatically by the runtime—that it is unreasonable to ask the programmer to determine when it is necessary. In either case, the question arises: how expensive must validation be?

With visible readers, validation is very inexpensive: a reader need only check to see whether it has been aborted. With invisible readers and eager acquire, naïve (incremental) validation takes time linear in the number of open objects. In a poster at PODC’04 [LM04], Lev and Moir suggested a heuristic that could reduce this cost in important cases. Specifically, they suggest per-object reader counters coupled with a global conflict counter. Readers increment and decrement the per-object counters at open and commit time, respectively. Writers increment the conflict counter whenever they acquire an object whose reader counter is nonzero. When opening a new object, a reader can skip incremental validation if the global conflict counter has not changed since the last time the reader checked it.
The conflict counter is a useful improvement over visible readers in systems like DSTM [HLMS03] and SXM [GHP05a], where visible readers require the installation of a new Locator and thus are very expensive. Unfortunately, every update of a reader counter will invalidate the counter in every other reader’s cache, leading to cache misses at commit time even when there are no writers.\footnote{With scalable nonzero indicators [ELLM07], cache misses can be avoided, though the overhead of atomic operations on semi-visible transactional reads remains [LLM+09].} In the absence of any contention, a transaction $T_1$ reading $R$ objects will skip all validation but must perform $2R$ atomic increment/decrement operations. For each object that is also read by $T_2$, $T_1$ will incur at least one cache miss, regardless of whether the counter is stored with the object metadata or in a separate cache line.

Validation can be omitted in the common case for invisible readers if the STM implementation uses timestamps [DSS06; RFF06; RFF07]. However, timestamp-based STM does not naturally provide privatization safety. It also complicates memory management: to free a region, timestamp-based STM algorithms require a transaction to appear to write every word of that region: If the metadata covering the entire region is not written, then subsequent transactional reads to the region by a doomed transaction will not result in an abort; instead those reads may return the value of nontransactional writes, if the region has been re-allocated to a nontransactional thread. This “sterilization” [DSS06] appears as a sequence of \texttt{CAS} instructions after committing, with the number of instructions linear in the size of the region being deleted. For languages without native array types, the programmer must know the size of any array being deleted, which may require changes to program code.

If one is willing to detect read-write conflicts lazily, a more orthogonal optimization can employ a global \textit{commit counter} that records only the number of writer transactions that have attempted to commit. When a transaction acquires an object, it sets a local flag
indicating that it must increment the counter before attempting to commit. Now when opening a new object, a reader can skip incremental validation if the global commit counter has not changed since the last time the reader checked it. If the counter has changed, the reader performs incremental validation.

In comparison to the Lev and Moir counter, this heuristic requires no atomic operations by readers, and the same amount of bookkeeping. A transaction $T_1$ that reads $R$ objects will validate by checking the global counter $R$ times. Reading the counter will be a cache miss only if a writing transaction commits during the execution of $T_1$, in which case an incremental validation is necessary. For a successful transaction $T_1$, the cost of validation with the global commit counter is a function of four variables: the number of objects read by $T_1$ ($R$), the number of writer transactions that commit during the execution of $T_1$ ($||\{T_w\}|| = W$), the cost of validating a single object (a cache hit and a single word comparison $C_v$, which we also use as the cost of detecting that the counter has not changed), and the cost of a cache miss ($C_{miss}$). Assuming that all $R$ objects fit in $T_1$’s cache, the baseline cost of incremental validation without the commit counter is $C_v \sum_{i=1}^{R} i = C_v \frac{R(R+1)}{2}$. Assuming a uniform distribution of writer commits across the duration of $T_1$, the cost of validation is the cost of $W$ successful validations of $R/2$ objects, $W$ cache misses, and $R - W$ successful checks of the global counter. For workload and machine configurations in which $C_v(R - W) + W(C_{miss} + \frac{C_v R}{2}) < C_v \frac{R(R+1)}{2}$, we expect the commit counter to offer an advantage in the common case. In the worst case, our mechanism will incur $\Omega(R^2)$ validation overhead. This cost is fundamental to opaque STM, even those algorithms that use timestamps [GK08].

**Mixed invalidation.** The global commit counter gets us partway to mixed invalidation: readers will notice conflicting writes only if (a) the writer acquires the object
before the reader opens it, or (b) some other writer transaction (which may not conflict with any readers) commits after the conflicting writer acquires and before the reader attempts to commit.

For comparison purposes, we have also built a full implementation of mixed invalidation. This implementation permits a transaction $T$ to read the old version of object $O$ even if $O$ has been acquired by transaction $S$, so long as $S$ has not committed. To correctly permit this “read through” operation, we augmented RSTM with a two-stage commit, similar to that employed by OSTM [Fra03; FH07]. A writer transaction $S$ that is ready to commit first CAS-es its status from active to finishing. $S$ then attempts to CAS the global commit counter to one more than the value $S$ saw when it last validated. If the increment fails, $S$ revalidates its read set and re-attempts the increment. If the increment succeeds, $S$ attempts to CAS its status from finishing to committed.

If transaction $T$ reads $O$, then when $S$ increments the counter, we are certain that $T$ will validate before accessing any new state; this preserves consistency. Furthermore, although $T$ can validate $O$ against the old version when acquirer $S$ is active, once $S$ changes its status to finishing and increments the counter, $T$ will fail validation.

To preserve non-blocking properties, any transaction can (with permission from the contention manager) abort $S$ even if it is finishing. In particular, if $T$’s validation fails and $T$ restarts, it will have the opportunity to abort $S$ if it tries to open $O$.

**Single thread optimization.** A transaction can easily count the number of times that it commits a writing transaction without ever needing incremental validation. If this occurs many times in succession, the thread can assume that it is running in isolation and skip all read-set bookkeeping and contention management calls (it must still increment the counter at the end of each write transaction). Should the global counter change due
to activity in another thread, such an opportunistic transaction will have to abort and retry.

Using this optimization, transactions with large read sets can skip the $O(n)$ time and space overhead of bookkeeping, resulting in significant speedup for single-threaded transactional code.

### 2.4 Experimental Evaluation of Conflict Detection and Validation Strategies

In this section we evaluate the effectiveness of six different conflict detection strategies. For comparison, we also plot results for coarse-grained locks and for the Lev and Moir conflict counter. We consider different lookup/insert/remove ratios for benchmarks that include a lookup operation, and show that as the read ratio increases, so does the relative benefit of the global commit counter. Thus while no single conflict detection strategy offers consistently superior performance, we believe that our approximation of mixed invalidation constitutes an important new point in the design space.

We also show that due to the cost of atomic operations on the critical path of every read, the Lev and Moir heuristic performs roughly at the level of visible readers in RSTM, rarely outperforming even the baseline RSTM system with invisible reads and eager acquire.

We performed all experiments on a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2GHz UltraSPARC III CPUs. All code was compiled with `GCC v3.4.4` using -O3 optimizations. For each benchmark and lookup/insert/remove mix, we averaged the throughput of three 10-second executions. For RSTM benchmarks, we used the `Polka` contention manager [SS05a].
2.4.1 Strategies Considered

RSTM supports both visible and invisible readers, and both eager and lazy acquire. We examine every combination other than visible reading with lazy acquire, which offers poor performance for our benchmarks and has comparatively weak motivation: while visibility allows readers to avoid incremental validation even when (unrelated) writers have committed, the effort they expend making themselves visible to writers is largely ignored, since writers delay conflict detection until commit time.

Visible readers with eager acquire (Vis-Eager) provides early detection of all conflicts without incremental validation. Invisible readers with eager acquire (Invis-Eager) also results in eager detection of all conflicts. Since reads are invisible, however, an acquiring transaction cannot detect that an object is being read; consequently, the acquirer cannot perform contention management but instead acquires the object obliviously, thereby implicitly dooming any extant invisible readers. To ensure consistency, transactions must incrementally validate their read set on every API call.

Invisible reads with lazy acquire (Invis-Lazy) results in lazy detection of all conflicts. This permits a high degree of concurrency among readers and writers, but requires incremental validation.

We also evaluate three heuristic validation methods, all based on a global commit counter.

In Invis-Eager + Heuristic, a transaction $T$ validates incrementally only if some writer transaction $W$ has incremented the counter since the last time $T$ validated. In addition to reducing the frequency of incremental validations, this permits some lazy detection of read-write conflicts. If $T$ reads $O$ and then $W$ acquires $O$, $T$ may still complete if no other writing transaction commits between when $W$ acquires $O$ and when $T$ commits.
Invis-lazy + Heuristic detects all conflicts lazily (at commit time). However, the heuristic permits a reduction in the overhead of validation: rather than incrementally validating on every API call, a transaction can validate trivially when no writer transaction $W$ has incremented the counter since the last time $T$ validated.

In Mixed Invalidation, read-write conflicts are detected lazily while write-write conflicts are detected eagerly. In contrast to Invis-Eager + Heuristic, Mixed Invalidation has precise conflict detection. For example, if $T$ reads $O$, then $S$ acquires $O$, then $W$ acquires some other object $P$ and commits, $T$ will not fail its validation; it will detect that $S$ has not committed, and that its version of $O$ is valid.

### 2.4.2 Benchmarks

We tested our conflict detection strategies against six microbenchmarks: a web cache simulation using least-frequently-used page replacement (LFUCache [SS04]), an adjacency list-based undirected graph (RandomGraph), and four variants of an integer set.

The LFUCache benchmark uses a large array-based index and a small priority queue to track frequently accessed pages in a simulated web cache. When the queue is reheapified, we introduce hysteresis by swapping value-one nodes with value-one children. This helps more pages to accumulate hits. A Zipf distribution determines the likelihood that a page is accessed, with probability of an access to page $i$ given as $p_c(i) \propto \sum_{0 \leq j \leq i} j^{-2}$.

In the RandomGraph benchmark, there is an even mix of inserts and deletes. When a node is inserted, it is given four randomly chosen neighbors. As nodes insert and leave the graph, the vertex set changes, as does the degree of each node. The graph is implemented as a sorted list of nodes, with each node owning a sorted list of neigh-
bors. Every transaction entails traversal of multiple lists; transactions tend to be quite complex. Transactions also tend to overlap significantly; it is rare to have an empty intersection of one transaction’s read set with another transaction’s write set.

In the integer set benchmarks, we consider an equal ratio, consisting of one-third each of lookup, insert, and remove operations, and a read-heavy mix with 80% lookups and 10% each inserts and removes.

The integer set benchmarks are a red-black tree, a hash table, and two sorted linked lists. Transactions in the hash table insert or remove one of 256 keys from a 256 bucket hash table with overflow chains. This implementation affords high concurrency with very rare conflicts. The red-black tree is a balanced binary tree of values in the range 0..65535. The linked lists hold values from 0..255; one list uses early release [HLMS03] to avoid false conflicts; the other does not.

### 2.4.3 Discussion of Results

In LFUCache (Figure 2.2), transactions usually do only a small amount of work, accessing one or two objects. Furthermore, the work done by all transactions tends to be on the same object or small set of objects. As a result, there is no significant parallelism in the benchmark. Lazy acquire performs best in this setting, because it shrinks the window of contention between two transactions, decreasing the chance that a transaction that successfully acquires an object will be aborted. Furthermore, since the read and write sets are small, the global commit counter saves little validation effort. The only benefit of our heuristic is slightly better performance in the single-threaded case.

RandomGraph (Figure 2.3), by contrast, benefits greatly from a global commit counter. Its transactions’ read sets typically contain hundreds of objects. Avoiding
incremental validation consequently enables orders of magnitude improvement. We observe real scalability with all three heuristic policies. This scalability is directly related to relaxing the detection of read-write conflicts: reading and acquiring are heavily interleaved in the benchmark, and detecting read-write conflicts early leads to near-livelock, as shown by the Invis/Eager line. Mixed invalidation, moreover, outperforms the best lazy conflict detection strategy. This is a direct consequence of avoiding concurrent execution of two transactions that want to modify the same object, a scenario we have previously identified as dangerous. The overhead of STM still gives coarse-grain locks a dramatic performance advantage, ranging from more than two orders of magnitude at low thread counts to a factor of almost 3 with 28 active threads.

The LinkedList benchmarks (Figures 2.4–2.5) show a tremendous benefit from the global commit counter when early release is not used, and a small constant improvement with early release. The difference stems from the fact that without early release
Figure 2.3: RandomGraph. For readability, coarse-grain locks are omitted; the curve descends smoothly from 250 KTx/sec at 1 thread to 52 KTx/sec at 28.

this benchmark is largely serial: the average reader opens 64 nodes to reach the middle of the list; any concurrent transaction that modifies an early node will force the reader to abort. With early release the programmer effectively certifies that modifications to early nodes are irrelevant once the reader has moved past them. No transaction keeps more than 3 nodes open at any given time, greatly increasing potential concurrency. Since transactions that modify the list do so with an acquire at the end of their transaction, there is little benefit to a relaxation of read-write conflict detection. The commit counter effectively reduces the frequency of incremental validation, however, and also significantly improves the single-threaded case.

In the RBTree benchmark (Figure 2.6), transactions tend to be small (fewer than 16 objects in the read set), with limited conflicts. As a result, decreasing the cost of validation does not significantly improve performance, nor does relaxing read-write conflict detection. However, the heuristic significantly improves the single-threaded
(a) 33% lookup, 33% insert, 33% remove. Coarse-grain locks descend smoothly from 1540 KTx/sec at 1 thread to 176 KTx/sec at 28.

(b) 80% lookup, 10% insert, 10% remove. Coarse-grain locks descend smoothly from 1900 KTx/sec at 1 thread to 328 KTx/sec at 28.

Figure 2.4: Sorted List Workloads
(a) 33% lookup, 33% insert, 33% remove. Coarse-grain locks descend from 1492 KTx/sec at 1 thread to 171 KTx/sec at 28.

(b) 80% lookup, 10% insert, 10% remove. Coarse-grain locks descend from 1994 KTx/sec at 1 thread to 354 KTx/sec at 28.

Figure 2.5: Sorted List Workloads with Early Release
case. The value of the heuristic also increases noticeably with the fraction of read-only transactions, as the cost of validation becomes a larger portion of overall execution time.

Unlike the other benchmarks, HashTable (Figure 2.7) is hurt by the global commit counter. The table is only 50% loaded on average, and with 256 buckets, the likelihood of two transactions conflicting is negligible. Furthermore, non-conflicting transactions do not read any common data objects. As a result, the benchmark is “embarrassingly concurrent.” The introduction of a global counter serializes all acquiring transactions at a single memory location, and thus decreases opportunities for parallelism. Some of this cost is regained with mixed invalidation, especially when there is a high percentage of read-only transactions.

2.5 Related Work

Marathe’s review of nonblocking STM implementations [MSS04] identified indirection and conflict detection (eager versus lazy) as key design parameters, and influenced the design of the ASTM [MSS05] and RSTM [MSH+06a] systems, which decrease overhead on the critical path of transactions. ASTM adaptively switches from DSTM-style eager acquire [HLMS03] to OSTM-style lazy acquire [Fra03; FH07]. This permits some dynamic determination of how and when transactions should validate, but it is not as nuanced as mixed invalidation and does not avoid unnecessary validation.

RSTM, added the ability to switch between visible and invisible readers on a per-object basis, though without automatic adaptation. RSTM thus subsumes the flexibility of Herlihy’s SXM [GHP05a], which uses a factory to set visibility for entire classes of objects. While visible readers offer potential gains in fairness by allowing contention
(a) 33% lookup, 33% insert, 33% remove. Single-threaded performance with coarse-grain locks is 2508 KTx/sec.

(b) 80% lookup, 10% insert, 10% remove. Single-threaded performance with coarse-grain locks is 3052 KTx/sec.

Figure 2.6: Red-Black Tree workloads
(a) 33% lookup, 33% insert, 33% remove.

(b) 80% lookup, 10% insert, 10% remove.

Figure 2.7: Hash Table
management for writes following uncommitted reads, we have found the cost in terms of reduced cache line sharing and reduced scalability to be unacceptably high; visible readers generally scale far worse than invisible readers when more than 4 threads are active. Even recent contention-free visible reader implementations [DS09b; LLM+09] incur some latency per read, due to either CAS instructions or explicit write-before-read memory fences.

Intel’s McRT-STM [SATH+06] uses locks to avoid the need for object cloning, thereby improving performance. The McRT compiler inserts periodic validation checks in transactions with internal loops, to avoid the performance risk of long-running doomed transactions. As in OSTM, the programmer must insert any validation checks that are needed for correctness.

Proposals from Microsoft Research [HMPH05; HPST06] implement transactions in Haskell and C#. The C# STM uses aggressive compiler optimization to reduce overheads, while the Haskell TM focuses on rich semantics for composability. Like previous word-based STM implementations [HF03; ST95], the Haskell system avoids the cost of copying unmodified portions of objects, but incurs bookkeeping costs on every load and store (or at least on every one that the compiler cannot prove is redundant). The C# system exclusively uses eager acquire. Although these differences complicate direct comparisons between word-based and object-based STM systems, we believe that our heuristic mixed invalidation would be a useful addition to word-based STM, and might assist developers in further reducing overheads while avoiding the problems of timestamps.

Several HTM proposals [AAK+05; HM93; HWC+04; MBM+06] seek to leverage cache coherence protocols to achieve lightweight hardware transactions. However, these hardware TMs generally fix the conflict detection policy at design time [BMV+07],
with eager read-write conflict detection more common than lazy [MBM+06]. To manage complexity, some of these HTM systems are best-effort only: for transactions with long running times or large read and write sets, the system falls back to a software-only mode [DFL+06; KCH+06; Moi05; MMN08]. Unfortunately, these approaches fix the conflict detection policy of both hardware and software.

There exist several proposals for hardware mechanisms to accelerate STM performance [MTC+07; SATJ06; SDS08; SMD+06b; SSH+07], of which Shriraman’s FlexTM [SDS08] allows adaptivity between eager and lazy acquisition. We believe this approach is more pragmatic: software dictates conflict detection and resolution policies, but special hardware instructions and cache states permit the small transactions in the common case to run as fast as coarse-grained locks.

The experiments in this chapter were conducted in 2006, and originally published at the same time as the first timestamp-based STM algorithms [DSS06; RFF06]. Since timestamp-based STM does not require polling on every read, the overhead is slightly less than our commit counter, particularly for small transactions or machines with high cache miss latencies. Experience with timestamp-based STM suggests that for low thread counts, the advantage over our commit counter is small. As the thread count increases, the costs of polling and excess validation decrease the appeal of our commit counter. However, these observations all depend on how timestamps are implemented. When timestamps are implemented pessimistically [DSS06], they can introduce significant extra aborts [DS07]; optimistic timestamp implementations [FFR08; RFF07] share a worst-case $\Omega(R^2)$ overhead with our mechanisms. In all cases, straightforward timestamp implementations complicate memory reclamation and are not privatization-safe. Making these systems privatization safe in unmanaged code appears to require our polling mechanism [DHMD07; DS09b; MSS08; SDMS08; SMDS07].
The only other orthogonal heuristic validation proposal we are aware of is the Lev and Moir conflict counter described in Section 2.3 [LM04]. While this heuristic removes unnecessary validation, it does not delay the detection of read-write conflicts. Inserting atomic operations into the critical path of every read shares lower-bound complexity with our visible reader implementation; we have shown that this strategy suffers the same costs (less cache line sharing, more processor stalls) as our visible reader implementation, and thus does not scale to high thread counts as well as invisible readers.

2.6 Conclusions

In this chapter, we presented a comprehensive and detailed analysis of conflict detection strategies. Using the original nonblocking RSTM algorithm, we assessed existing policies for detecting and resolving read-write and write-write conflicts using reader visibility and acquire time, and discussed the utility of mixed invalidation in avoiding conservative aborts of transactions that may be able to succeed.

Mixed invalidation can be approximated in eager STM using a global commit counter heuristic. Our implementation is orthogonal to STM design, and demonstrates that the resulting gain in concurrency can lead to significant performance improvements in workloads with long, highly contended transactions. The global commit counter can also be used to detect the case when a system contains only one transactional thread, which can then opportunistically avoid the overhead of bookkeeping and contention management.

Our heuristics are still insufficient to close the performance gap between STM and locks in all cases. In fact, the global commit counter can decrease performance in highly concurrent workloads (such as hash tables) by forcing all transactions to serialize...
on a single memory location when they otherwise would access disjoint memory sets. Nonetheless, mixed invalidation appears to be a valuable step toward maximizing STM performance. Hardware acceleration of polling operations [SSD+07] may help to close the performance gap.

The fact that no one conflict detection or validation mechanism performs best across all workloads—and that the differences among mechanisms are both large and bidirectional—suggests that a production quality STM system should adapt its policy to match the offered workload. The ASTM system [MSS05] adapted in some cases between eager and lazy acquire; more adaptation is clearly worth exploring.
3 Lowering The Latency Of Software Transactional Memory

In Section 1.3.1, we considered an approach to STM based on ownership records (orecs), in which metadata is attached to ranges of shared memory. Subsequent runtime instructions read and update this metadata in order to ensure that an in-flight transaction’s reads and writes remain correct. The overhead of metadata manipulation and inspection is linear in the number of reads and writes performed by a transaction, and involves one read-modify-write instruction per location written by the transaction, resulting in substantial overheads. The addition of a commit counter to this design, as in Chapter 2, can avoid much overhead in the common case, but substantial per-access and commit-time overheads remain. STM implementations may never be “fast enough” when metadata manipulation overheads are linear in the amount of work done.

In this chapter we present an algorithm in which transactions represent their read and write sets as Bloom filters, and transactions commit by enqueuing a Bloom filter onto a global list. Using this approach, our RingSTM system requires at most one read-modify-write operation for any transaction, and incurs validation overhead linear not in transaction size, but in the number of concurrent writers that commit. Furthermore, RingSTM is the first STM that is inherently livelock-free and privatization-safe while
at the same time permitting parallel writeback by concurrent disjoint transactions. We evaluate three variants of the RingSTM algorithm, and find that it performs competitively with other STM algorithms, despite also providing these stronger progress and privatization safety guarantees.

### 3.1 Costs and Benefits of Orec STM

The recent flurry of research into the design and implementation of high-performance blocking and nonblocking STM algorithms [DS09b; DSS06; Fra03; FH07; GHP05a; HF03; HLMS03; HPST06; LLM+09; MSH+06a; MSS05; SATH+06] was primarily influenced by Shavit and Touitou’s original STM [ST95]. In this design, transactional data is versioned explicitly by associating “ownership records” (orecs) with ranges of shared memory, where each orec encapsulates version and ownership information. The Shavit and Touitou STM has formed the basis of numerous successful and scalable runtimes, most notably TL2 [DSS06], widely considered to be the state-of-the-art.

#### 3.1.1 Tradeoffs in Orec-Based STM

In orec-based runtimes such as TL2, the use of per-location metadata represents a tradeoff between single-thread overhead and scalability. The main overheads are outlined below; latency is incurred on any metadata access. In return for these costs, transactions whose metadata accesses do not conflict can commit concurrently.

- Linear Atomic Operation Overhead: A transaction writing to \( W \) distinct locations must update the metadata associated with those \( W \) locations using atomic RMW operations such as `CAS`. Some implementations require atomic acquire
and atomic release, resulting in a total of $2W$ RMW operations, but even the fastest runtimes cannot avoid an $O(W)$ acquire cost. As we discussed in Section 1.1.1, each RMW operation has a high overhead, as it introduces some amount of bus traffic and ordering within the processor memory consistency model.

- **Linear Commit-Time Overhead:** In most STM algorithms, a committing writer incurs overhead linear in both the number of reads and the number of writes. The commit step of STM using commit-time locking consists of an $O(W)$ orecc acquisition phase, $O(R)$ read-set validation phase, $O(W)$ write-back phase, and $O(W)$ orecc release phase. When encounter-time locking is used, the $O(W)$ acquisition overhead is incurred during transaction execution, rather than at commit, and $O(W)$ write-back can be avoided. Furthermore, when writing transactions are infrequent, the final $O(R)$ validation overhead may be avoided via a commit counter [SMSS06] (as also discussed in Chapter 2) or global timestamps [DSS06; RFF06; ZBS08]. Several hardware proposals can decrease commit overhead to as low as $O(W)$ [MTC+07; SATJ06; SSD+07; SSH+07] (in the case of FlexTM, the overhead is linear in the number of writes and the number of concurrent transactions [SDS08]), albeit at increased hardware complexity. On existing hardware with a heterogeneous workload including a modest number of writing transactions, commit overhead (excluding acquisition) has a time overhead of $O(R + W)$, with high constants associated with the $W$ term.

- **Privatization Overhead:** When a transaction removes or otherwise logically disables all global references to a region of shared memory, subsequent access to that region should not require transactional instrumentation. However, such “privatizing” transactions may need to block at their commit point, both to permit
post-commit cleanup by logically earlier transactions, and to prevent inconsistent reads by doomed transactions [LR07; SMDS07]. While future research may discover efficient solutions to privatization, the current state of the art appears to involve significant overhead, with transactions blocking after commit in order to wait for all in-flight transactions to complete or abort and clean up.

- **Read Consistency Overhead:** A transaction reading $R$ distinct locations must ensure that there exists a logical time at which all $R$ reads were simultaneously valid. As discussed in Chapter 2, early nonblocking STM algorithms (such as DSTM [HLMS03], ASTM [MSS05], and RSTM [MSH+06]) achieve this condition through $O(R^2)$ incremental validation, which can be reduced via heuristics [SMSS06]; time-based STM algorithms [DSS06; RFF06; RFF07; WCW+07] use a postvalidation step that has constant overhead for each transactional read. Some STM algorithms ignore read consistency until commit-time, and rely on sandboxing, exceptions, and signals to detect inconsistencies [Enn06; Fra03; FH07; HPST06; SATH+06]. In these systems, additional instructions or compiler instrumentation may be required to detect infinite loops, and “doomed” transactions may run for extended periods (as long as a garbage collection epoch) after performing an inconsistent read.

### 3.1.2 Alternative Hardware and Software TMs

Among software-only TMs, the most prominent exception to the Shavit and Touitou design is JudoSTM [OCS07], which includes a variant in which all concurrency control is provided via a single global counter, which requires only one RMW operation per writer transaction. By using a single counter, sandboxing, and write-back on commit, JudoSTM implicitly avoids the privatization problem, and incurs only $O(1)$ atomic
operations and $O(R)$ commit-time validation overhead, for a total commit overhead of $O(W + R)$. However, these benefits come at the cost of strongly serialized commit: when a transaction is in the process of committing updates to shared memory, no concurrent reading or writing transactions can commit. In addition, JudoSTM must log twice as much information as TL2 in order to perform its conflict detection and resolution, since it must log both the location and the value it read. We have found that this cost can increase latency by 40% on the simple cores of the Sun Niagara processor [KAO05].

The only other oreo-free STM algorithm is DracoSTM [GC08]. DracoSTM uses a commit token, which blocks all transactions whenever any transaction wishes to commit. It does not require the extra logging overhead of JudoSTM, but requires all per-thread metadata to be visible to all committing transactions, which can introduce latency whenever a transaction performs a read, since it must update global metadata.

Several proposals for hardware transactional memory (HTM) do not require the use of per-location metadata, instead relying on cache coherence messages or custom processor data structures to ensure atomic and isolated updates by successful transactions [LR07]. Naturally, these systems incur minimal overhead (often constant). Unfortunately, these designs are either complex, or else limited in their ability to support transactions unbounded in space and time. Recent designs have shown, however, that Bloom filters [Blo70] can be implemented efficiently in hardware and used to accelerate memory transactions [CTTC06; MTC07; RHL05; YBM07].

### 3.1.3 Contributions

In this chapter we present RingSTM, a novel STM algorithm with several advantageous characteristics. By using per-transaction Bloom filters [Blo70] to detect con-
flicts among transactions, RingSTM avoids $O(R)$ commit-time overhead and lowers the constants associated with $O(W)$ commit-time overhead, providing much lower latency than orec-based STM designs. RingSTM also provides stronger and simpler semantics, and enables all of the following advantages at the same time: (1) read-write transactions commit using only one atomic operation, and read-only transactions do not require any atomic operations, (2) the runtime is inherently livelock-free, (3) non-conflicting read-write transactions do not prevent read-only transactions from committing, (4) all transactions are seamlessly privatization-safe, (5) non-conflicting read-write transactions can commit concurrently, and (6) the runtime avoids the space overheads of read set logging.

We consider three variants of RingSTM, which trade worst-case overheads for increasingly concurrent commit. Experiments on a 32-thread Sun Niagara T1000 chip multiprocessor show that RingSTM performs comparably to orec-based STM implementations. Based on these results, we believe that RingSTM is the preferred choice of STM design for a number of cases, such as when privatization is frequent, seamless privatization is necessary, writing transactions are rare, or transactions have very large read and write sets. When transactions are used to implement thread-level speculation [BMT+07; vPCC07], we also expect RingSTM to outperform other STM algorithms, due to its short commit sequence.

We describe the design of RingSTM in Section 3.2, and present three variants that trade higher concurrency for weaker worst-case guarantees on validation overhead in Section 3.3. Section 3.4 evaluates the performance of RingSTM, comparing to TL2. Lastly, we discuss future directions in Section 3.5.
3.2 RingSTM Design

RingSTM transactions detect conflicts and ensure atomicity, isolation, and consistency through manipulation of an ordered, fixed size ring data structure (the ring).\textsuperscript{1} Transactional writes are buffered and performed upon successful commit. Appending an entry to the ring effects a logical commit and is the explicit linearization point of a writing transaction; this ensures that ring entries describe only successful writer transactions, preventing unnecessary validation. In the default RingSTM configuration, the position of an entry in the ring specifies both logical and physical commit order (that is, when the transaction added a ring entry and when it completed write-back). This framework permits numerous novel design aspects, outlined below.

3.2.1 Global Metadata Describes Committed Transactions

Each entry in the fixed-size ring describes a successful writer transaction, and consists of four fields: a logical timestamp ($ts$), a Bloom filter representing the set of locations modified ($wf$), a status field, and a priority ($prio$). Figure 3.1 depicts an 8-entry ring. Only writing transactions update the ring, and entries are added to the ring with a single RMW operation. Initially, an entry’s status is writing, indicating that the corresponding transaction is logically complete, but it has not performed all of its writes. Once all writes are performed, the status is updated to complete. Multiple ring entries can be in the writing state.

When an entry’s status is writing, all newer transactions must treat the entry’s write set as locked. For concurrent committing writers, this means that write-after-write (WAW) ordering must be respected. We provide WAW between concurrent writers $T_1$

\textsuperscript{1}The semantic limitations imposed by a fixed-size ring are discussed in Section 3.3.4
Figure 3.1: Example ring holding 8 entries. The newest entry (47) represents a transaction that has linearized but has not started writeback. Entries 44...46 are performing write-back. Entries 40...43 are complete.

and $T_2$ as follows: If $T_1$’s ring entry has an earlier commit time than $T_2$’s entry, and $T_2$ determines that its write filter has a nonzero intersection with $T_1$’s filter, then $T_2$ cannot perform its writes until $T_1$ sets its entry’s status to complete. The update of $T_1$’s status serves to release the set of logical locks covering its write set.

The rest of the commit sequence has minimal overhead: the ring entry is initialized and added to the ring with a single RMW operation ($O(1)$). $W$ writes are then performed ($O(W)$), and the entry status is set to complete ($O(1)$). In contrast, oreo-based STM incurs $O(W)$ overhead to acquire and release up to $W$ physical locks, and $O(W)$ overhead to perform their writes.
3.2.2 Trading Precision for Speed in Validation

When a transaction $T_a$ begins, it identifies the oldest ring entry with a status of writing, and sets its start time to immediately before that transaction’s commit time. Whenever a shared location is read, $T_a$ validates by scanning the ring and identifying any entries with commit times later than its start time; $T_a$ intersects its read set with each such entry’s write set. If the intersection is nonzero, $T_a$ aborts and restarts. When $T_a$ computes a zero intersection with a filter $F$ whose writeback status is complete, $T_a$ can ignore all future intersections with $F$. Thus if $N_W$ transactions commit after $T_a$ begins, then $T_a$ can issue at most $R \times N_W$ intersections, but at least $N_W$ intersections, where $R$ is the number of locations read by $T_a$. For a fixed-size filter, this results in $\Omega(N_W)$ validation overhead.

There are a number of tradeoffs involved in validation. First, filter intersection is prone to false conflicts. If transactions logged their read sets precisely, they could test individual reads for membership in published write filters. However, such tests still admit false positives due to hash collisions, and we believe that adapting filter size while incurring only constant storage overhead is preferable to incurring linear space overhead and potentially $O(R^2)$ time overhead. Furthermore, when membership tests are not necessary, using fewer hash functions decreases false conflicts at any filter size. Bloom filters are also prone to the “Birthday Paradox” observation [ZR07], which can be mitigated to a degree by dynamically changing filter size.

Next, we note that the validation strategies listed above ensure consistent reads (no sandboxing) but introduce polling. Since each of the $N_W$ filters must be checked before $T_a$ commits, there is no practical benefit to replacing polling with sandboxing. In fact, polling decreases the incidence of false conflicts, since write filters are tested early and then ignored once their status is complete. If all potentially conflicting write filters
were tested during $T_a$’s commit, $T_a$’s read filter would be at its most full, increasing the chance that a read-after-write order be misinterpreted as a conflict. Furthermore, polling decreases the risk of a transaction experiencing ring overflow. If the number of write filters that must be validated exceeds the number of ring entries, then $T_a$ must abort and restart. By polling, $T_a$ can ignore logically older filters once they are complete, and such filters can then be reused by newer committing writers without impeding $T_a$’s progress.

3.2.3 Low Commit Overhead and Livelock Freedom

Transactions do not perform validation after acquiring a ring entry: instead, they validate before acquiring a ring entry and then attempt to add a ring entry by issuing a RMW operation that will only succeed if the ring is unchanged since that validation. This decision ensures that all ring entries correspond to committed transactions; that is, failed transactions never cause other transactions to fail. This decreases the incidence of ring overflow. Furthermore, since the creation of a ring entry logically locks all locations in the entry’s write filter, eliminating validation after issuing an RMW operation decreases the size of the commit critical section. This is a notable difference from oreo-based STM, where a writer must validate after locking, incurring $O(R + W)$ overhead in its critical section.

RingSTM is inherently lazy: until a transaction is ready to commit, its existence is not visible to concurrent threads; during the commit phase, all speculative writes must be replayed from a buffer after the transaction has become visible. Since the entire write set is locked via a single atomic operation, there is no need for contention management [SS05a] among incomplete transactions for livelock prevention: until a transaction is logically committed, it is invisible to other transactions; once a trans-
action is visible, it is represented by a ring entry and is logically committed. Thus
RingSTM is livelock free. Informally, a transaction $T_a$ will abort only if an entry $E$ is
added to the ring after $T_a$’s logical begin time, and $E$’s write filter has a nonzero inter-
section with $T_a$’s read filter. The addition of $E$ to the ring by transaction $T_e$ atomically transitions $T_e$ to a state in which it is logically committed. Since $T_a$ cannot be aborted by an incomplete transaction, $T_a$’s abort signifies that some transaction in the system (in this case $T_e$) made progress.

3.2.4 Contention Management

Bobba et al. warn that three “pathologies” affect transactional runtimes that use
defered updates [BMV+07]: commits can be unnecessarily serialized, starvation is possible, and multiple restarts can lead to convoying. They propose a minimal amount of bookkeeping to perform exponential backoff on abort, and show that it substantially reduces convoying. Since RingSTM only serializes the commits of transactions with likely WAW ordering constraints, we do not expect serialized commit to be a problem (Bobba et al. focus on the case where hardware TM uses a single commit token).

To address starvation, we maintain a priority field in each ring entry, and make use of a global priority manager. The priority manager handles requests to raise and lower the ring priority; writers can commit if and only if their priority matches the current ring priority, represented by the value of the priority field in the newest ring entry. In Chapter 5, we will present more scalable techniques that allow nonconflicting low-priority writers to commit, albeit at the cost of higher latency for the high-priority transaction.

If a transaction $T$ aborts repeatedly, it can request a higher priority $p'$ from the
global priority manager. If the request is granted, $T$ then commits a “dummy” trans-
action (represented by an empty write filter) with the elevated priority, and assumes responsibility for restoring the lower ring priority when it commits. As soon as the dummy transaction commits, a concurrent writing transaction $T_l$ of lower priority is effectively scheduled behind $T$ (it cannot commit until $T$ commits and lowers the ring priority), unless it also requests and receives elevated priority from the priority manager. Since $T_l$ cannot commit, it is incapable of changing shared memory and thus it cannot cause $T$ to abort.

After raising the ring priority, $T$ continues, and when it is ready to commit, it does so by adding a ring entry describing its write set. This entry also restores the original ring priority, and thus as soon as $T$ commits, all lower-priority writer transactions can attempt to commit. If $T$ is a high-priority read-only transaction, it must still commit an entry with an empty write filter that restores the ring priority.

If some other transaction $T'$ also receives elevated priority from the global priority manager, many interleavings are possible. If $T'$’s priority matches $T$’s priority, then $T'$ can commit first with no complications. However, if $T$ commits first and restores the old ring priority, then $T'$ will detect that the ring has lost its elevated priority when it validates, and must re-raise the priority. When $T'$ commits in this setting, it must re-lower the priority as well. If $T'$ receives higher priority than $T$, it immediately commits a dummy transaction with elevated priority, in which case $T$ will be scheduled behind $T'$, and $T_l$ will still be scheduled behind $T$. When $T'$ commits, it restores the ring to $T$’s priority, and when $T$ commits, it restores the ring to its original priority.

The priority field of a ring entry is orthogonal to the write filter of that entry, and thus low-priority read-only transactions are never blocked by high-priority transactions: First, low-priority read-only transactions will never modify the ring, and thus are not blocked by the requirement that new ring entries cannot lower the ring priority unless
they had previously committed a dummy transaction to raise the priority. Secondly, read-only transactions linearize at their final validation, which is dependent only on the write filters of ring entries, not the priority fields.

If starvation persists (due to multiple transactions executing at a higher priority), \( T \) may request permission to become inevitable [BDLM07; BZ07; HPR+07; OCS07; SMS08]. In RingSTM, a transaction becomes inevitable by adding a writing entry with a full filter (all bits set) with status writing, and then waiting for all older ring entries to set their status to complete. It then performs all writes in-place, and at commit time, it updates the status of its entry to complete. Inevitable transactions forbid all concurrency and limit the use of retry-based condition synchronization, but can call most precompiled libraries and perform I/O.

3.2.5 Privatization is Free

In its default configuration, RingSTM does not incur any overhead to support privatization. We argue this position by demonstrating that neither “half” of the privatization problem [SMDS07] (Section 1.5.3) affects RingSTM.

The “deferred update” problem deals with nontransactional code failing to observe delayed transactional writes to a privatized region. Since ring entries are in logical commit order, it is necessary only to ensure that all logically older transactions than a privatizing transaction \( T_p \) have set their status to complete before \( T_p \) performs uninstrumented reads to newly privatized memory. Two of the three variants of RingSTM presented in Section 3.3 give this guarantee implicitly; the third can provide this guarantee by spinning on a single global variable, as discussed in Section 3.3.3.

The “doomed transaction” problem requires the runtime to prevent any transaction \( T_r \) that conflicts with \( T_p \) from observing private writes that occur after \( T_p \) commits.
but before \( T_r \) aborts. Since RingSTM transactions poll the ring on every transactional read, and validate their entire read set through filter intersections whenever the ring is modified, there is no doomed transaction problem. \( T_r \) will detect the conflict on its first read after \( T_p \) commits, and will abort immediately, without using potentially inconsistent values.

### 3.2.6 Minimal Memory Management Overhead

Many STM algorithms rely on garbage collection or epoch-based deferred reclamation, and consequently suffer from heap blow-up when there is any concurrency [Fra03; FH07; HSATH06]. TL2 avoids this requirement by making a call to `free` appear to write to the entire region being freed (achieved by updating timestamps in all orecs associated with memory to be freed). The worst-case cost of this “sterilization” process is a number of RMW operations proportional to the size of the region being freed.

Sterilization in TL2 is necessary because TL2 is not privatization-safe. If \( T_1 \) is reading a region of memory that \( T_2 \) frees, and the memory manager recycles the region and passes it to a nontransactional thread \( Q \), then \( Q \)’s subsequent uninstrumented writes will be visible to \( T_1 \), which may fail to detect that it must abort due to conflict with a committed \( T_2 \). By sterilizing metadata before freeing memory, TL2 ensures that doomed readers of a freed region abort.

Since RingSTM transactions poll for conflicts, and validate their entire read set whenever a conflict is possible, there is no need for sterilization or any other special mechanism. However, as in TL2, non-faulting loads (or the appropriate signal handlers) may be required if the underlying allocator returns memory to the operating system while it is being read by a concurrent (doomed) transaction, and `malloc` and `free`
struct RingEntry
    int ts // commit timestamp
    filter wf // write filter
    int prio // priority
    int st // writing or complete

struct Transaction
    set wset // speculative writes
    filter wf // addresses to write
    filter rf // addresses read
    int start // logical start time

RingEntry ring[] // the global ring
int ring_index // newest ring entry
int prefix_index // RingR prefix field

Listing 3.1: RingSTM metadata

instructions must be logged, with frees deferred until commit and mallocs undone upon abort.

3.3 Implementation

In this section we present three variants of the RingSTM algorithm. The default version (RingSTM) permits parallel writeback and incurs no privatization overhead. The single-writer algorithm (RingSW) restricts the runtime to a single committing writer at a time, but offers low asymptotic validation overhead. The relaxed writeback algorithm (RingR) relaxes the writeback mechanism relative to the default algorithm, at the expense of additional overhead for privatization. The global metadata describing transactions and ring entries are common to all algorithms, and appear in Listing 3.1. The ring is initialized so that ring_index == 0 and every RingEntry has ts == 0 and st == complete.
tm_read:
1 if addr in TX.wf && addr in TX.wset
2 return lookup(addr, TX.wset)
3 val = *addr
4 TX.rf.add(addr)
5 read-read-fence
6 check()
7 return val

tm_write:
1 TX.wset.add(addr, val)
2 TX.wf.add(addr)

Listing 3.2: Read and write instrumentation, common to all RingSTM variants.

3.3.1 The Default RingSTM Algorithm

Pseudocode for the instrumentation required on each read and write in RingSTM appears in Listing 3.2. Boundary instrumentation, and the validation operation, appear in Listing 3.3; for clarity, the pseudocode assumes an infinite-size ring and omits constant-time ring overflow tests and modular arithmetic to compute ring indices from timestamps. Ring rollover is discussed in Section 3.3.4. The global ring data structure stores RingEntry records, as described in Section 3.2.1, and the global integer ring_index stores the timestamp of the newest ring entry. Transaction is a private per-transaction descriptor storing Bloom filters representing the addresses read (rf) and written (wf), and a buffer of speculative writes (wset). The start field holds the timestamp of the last writer to complete before this transaction began.

The foundation of the default algorithm is an invariant describing ring entries. Let $L_i$ represent the $i$th entry in the ring, with larger values of $i$ indicating newer entries in the ring. We define the suffix property as:

$$L_i.st = \text{writing} \implies \forall_{k>i} L_k.st = \text{writing}$$ (3.1)
Listing 3.3: Pseudocode for RingSTM (default algorithm) begin, end, and validate functions.

Informally, the suffix property ensures that committing transactions release the logical locks covering their write set in the same order as the transactions logically committed (acquired entries in the ring). For example, in Figure 3.1, entries {44... 47} com-
prise the suffix, and entry 45 cannot set its status to complete while entry 44 is still writing.

Equation 3.1 enforces ordered completion among writers, and prevents the “delayed update” half of the privatization problem: If $T_2$ privatizes a region with outstanding updates by logically earlier transaction $T_1$, then $T_2$ will not exit its commit sequence until $T_1$ completes its writeback and releases its logical locks; hence $T_2$ is guaranteed to observe $T_1$’s writes. When $T_1$ and $T_2$’s write sets do not overlap, the corresponding physical writes can occur in any order; suffix-based serialization only applies to the order in which ring entries are marked complete.

Transactions set their start field via an $O(N_W)$ call to $tm\_begin$, which computes start time by determining the oldest entry $E$ in the ring with incomplete writeback. By choosing a start time immediately before this $E$’s completion, a new transaction is certain not to overlook pending writeback by logically committed transactions, without having to wait for that writeback to complete.

Transactional writes are buffered in a log ($TX\_wset$); the address to be written is also added to a write filter $TX\_wf$. To read a location, the transaction checks the log, and immediately returns when a buffered write is present. Otherwise, the transaction reads directly from memory, adds the location to its read filter ($TX\_rf$), and then polls for conflicts via the check function. Alternative implementations are possible that do not update $rf$ until after calling check, thereby avoiding aborts when a read to a new location conflicts with a ring entry that is writing.

The check function validates a transaction against every writer that committed after the transaction’s logical start time. When it encounters a committed writer ($T_c$) that is no longer writing, the check function resets the transaction’s start time to after $T_c$. In this manner, subsequent reads to locations that were modified by $T_c$ will
not result in conflicts. Since each call to check can lead to as many as $N_W$ filter intersections (where $N_W$ is the number of committing writers during a transaction’s execution), the maximum validation overhead is $(R + 1) \times N_W$. In practice, we expect this overhead to tend toward its lower bound of $\Omega(N_W)$.

Lastly, as discussed in Section 3.2.3, $tm$\_end ensures a minimal contention window by performing all validation before acquiring a ring entry (and hence locking its write set). On line 6, a write-before-write fence may be needed to ensure that the $wf$ and $st$ fields are written before the $ts$ field. The $O(N_W)$ loop on lines 7-9 ensures that write-after-write ordering is preserved, but otherwise writeback (lines 11-12) can occur in parallel. Lines 13-15 preserve the suffix property by ensuring that entries are marked complete in order.

As noted in Section 3.2.5, this Ring algorithm is privatization-safe. To protect doomed transactions, every transaction polls for conflicts before using the value of any transactional read, ensuring that transactions abort before using inconsistent values. By ensuring that writer transactions depart the $tm$\_end function in the order that they logically commit, there is no risk of committing and then reading stale values due to delayed writeback by an older transaction.

### 3.3.2 The Single Writer Algorithm (RingSW)

For workloads with high proportions of read-only transactions, the value of parallel writeback can be outweighed by the $R$ factor in the worst-case overhead of validation. The single-writer variant of the RingSTM algorithm (RingSW) reduces validation overhead to exactly $N_W$ filter intersections, but forbids concurrent writeback. In effect, RingSW is an optimized version of the default algorithm when the suffix is bounded to a length of at most one. Listing 3.4 presents pseudocode for RingSW.
tm_begin:
1 read ring_index to TX.start
2 if ring[TX.start].st != complete ||
   ring[TX.start].ts < TX.start
3 TX.start--
4 read-read-fence

tm_end:
1 if read-only return
2 commit_time=ring_index
3 check()
4 if !CAS(ring_index, commit_time, commit_time+1)
5 goto 2
6 ring[commit_time+1] = (writing, TX.wf, commit_time+1)
7 write-write-fence
8 for (addr, val) in TX.wset
9   *addr = val
10 write-write-fence
11 ring[commit_time+1].st=complete

check:
1 my_index=ring_index
2 if my_index == TX.start return
3 while (ring[my_index].ts < my_index) SPIN
4 read-read-fence
5 for i = my_index downto TX.start+1
6   if intersect(ring[i].wf, TX.rf)
7     abort()
8 while ring[my_index].st != complete
9   SPIN
10 read-read-fence
11 TX.start = my_index

Listing 3.4: Pseudocode for RingSW

The principal differences between the default algorithm and RingSW are that (1) in RingSW, tm_begin() has constant overhead, and (2) in RingSW, tm_end both skips write-after-write ordering and ignores the suffix property, since it is implicit in the single-writer protocol. All enforcement of the single-writer protocol is performed in the check function, via lines 7-9. In addition to blocking (which should not dominate
as long as $R \gg N_W$), enforcement of the single-writer protocol introduces at most $N_W$ read-before-read memory fences on processors with relaxed memory models, to order the spin on line 8 before the next call to `tm_read`.

As a specialized implementation of the default algorithm for suffix lengths no greater than 1, RingSW has no privatization overhead. The main differences deal with serialization and complexity: RingSW forbids concurrent writeback by nonconflicting transactions, but has $O(1)$ overhead in `tm_begin`, exactly $N_W$ validation overhead, and only $O(W)$ overhead in `tm_end`. In contrast, the default algorithm has a worst-case $O(N_W)$ overhead in `tm_begin` and `tm_end`.

### 3.3.3 Relaxed Commit Order (RingR)

While the suffix property specified in Equation 3.1 permits concurrent commit, it enforces ordered departure from `tm_end` for writing transactions. We now present the relaxed commit order algorithm (RingR), which permits disjoint writers to complete write-back in any order and immediately execute new transactions. In return for this relaxation, RingR transactions are not privatization-safe.

In the default Ring algorithm, the suffix property implies a prefix property as well. If $L_i$ represents the $i$th entry in the ring, then:

$$\exists i: L_i.st = complete \implies \forall k \leq i L_k.st = complete$$

Informally, the prefix property states that regardless of the state of the newest entries in the ring, the set of “oldest” entries in the ring have all completed writeback. In the default algorithm, the suffix and prefix are adjacent, and their union includes all ring entries. In RingR, we preserve the prefix property but not the suffix property. Since
there can be multiple ring entries that satisfy the prefix property, we safely approximate
the prefix with the prefix\_index global variable. At times, prefix\_index may
not store the largest value of $i$ for which Equation 3.2 holds, but it will always describe
a correct prefix of the ring.

Pseudocode for RingR appears in Listing 3.5. The principal difference is in the
maintenance of the prefix\_index variable. As in the default, successful calls to
check advance the logical start time of the transaction; however, we now use this start
time to assist in maintaining the prefix\_index. Since the logical start time is based
on the oldest committed transaction for which writeback is incomplete, the transaction
that committed immediately before a transaction’s logical start time is a valid value of
$i$ for Equation 3.2.

Since there are many correct values for the prefix\_index, we maintain its value
through regular stores (as opposed to atomic instructions). Occasionally this will result
in the prefix moving “backward”, but does not compromise correctness. Furthermore,
even though readers may be aware of a more precise value of prefix\_index, only
successful writers maintain the variable. This ensures that read-only transactions never
modify global metadata. Use of prefix\_index is straightforward: in \texttt{tm\_begin}, the
transaction reads prefix\_index and then advances its start time as far as possible.
For a bounded ring, the overhead of this loop is linear in ring size; in practice, we expect
the prefix\_index to be close to optimal, introducing little common-case overhead.

Since writers can exit \texttt{tm\_end} in any order, RingR admits a deferred update prob-
lem. To make the algorithm privatization-safe, we can ensure that a committed priva-
tizer $T_p$ does not access shared data nontransactionally until prefix\_index $\geq T_p$.commit\_time. Once the prefix\_index advances to $T_p$’s commit time, $T_p$ is guaranteed
that there are no deferred updates to the privatized region. Since transactions still poll for conflicts, RingR does not introduce a doomed transaction problem.

3.3.4 Ring Rollover

To support a bounded ring, only a small number of constant-overhead instructions are needed. First, all ring indices must be computed via modular arithmetic, and whenever the expected timestamp of an entry is greater than the value expected by a trans-
action (either during check or tm_end), that transaction must abort. Secondly, when check is called, a rollover test must be issued before returning, to ensure that, after all validation is complete, the oldest entry validated has not been overwritten (accomplished with a test on its timestamp field). This test detects when a validation is interrupted by a context switch, and conservatively aborts if ring rollover occurs while the transaction is swapped out. When RingSTM is implemented on processors with relaxed memory consistency, the preemption test will need to be ordered via a read-read memory fence. Lastly, the order in which ring entries are updated matters. During initialization, updates to a new ring entry’s timestamp must follow the setting of the write filter and the status (write-after-write ordering). Calls to check from tm_end must block until the newest ring entry’s timestamp is set.

RingSW requires no further changes. In the default RingSTM algorithm, threads must ensure that there is always at least one ring entry whose status is writeback complete. When the number of active transactions is less than the number of ring entries, this requirement is implicitly satisfied. Otherwise, a test is required in tm_end before acquiring a new ring entry. This test is required in RingR, which must also ensure that the prefix_head always points to a live entry in the ring. This can be achieved either through the use of an atomic RMW instruction, or by adding an additional test to the use of prefix_head in tm_begin.

3.3.5 Summary

In each of the RingSTM implementations, only one RMW operation is required, regardless of write set size. In the common case, all algorithms should incur overhead linear only in $N_W$, the number of concurrent writers. Privatization is either free, or can be achieved by waiting for a single global variable to satisfy a simple predicate. Read-
only transactions commit without issuing writes, and read-only workloads can avoid all validation, although they must issue \( R \) tests of the `ring_index` variable.

### 3.4 Evaluation

We implemented the RingSTM, RingSW, and RingR algorithms as a C library, compatible with 32-bit C and C++ applications on the POWER, SPARC, and x86 architectures. In this section we focus on the results on an 8-core (32-thread), 1.0 GHz Sun T1000 (Niagara) chip multiprocessor running Solaris 10; we believe the Niagara is representative of a strong emerging trend in processor design. All benchmarks are written in C++ and compiled with g++ version 4.1.1 using –O3 optimizations. Each data point is the median of five trials, each of which was run for five seconds.

#### 3.4.1 Runtime Systems Evaluated

We parameterized each of the RingSTM algorithms by filter size, using small (32-bit), medium (1024-bit), and large (8192-bit) filters. The medium and large filters are optimized through the addition of a 32-bit summary filter; filter intersections are computed on the summary filters first, with the larger filters used only when the summary intersection is nonzero. In all runtimes, the ring stores 1024 entries.

We compare the resulting nine RingSTM systems against a local implementation of the published TL2 algorithm [DSS06], which represents the state of the art. Our implementation of TL2 uses an array of 1M ownership records, and resolves conflicts using a simple, blocking contention management policy (abort on conflict). Our implementation of TL2 shares as much code as possible with RingSTM, to prevent implementation artifacts from affecting results.
Figure 3.2: Red-black tree storing 20-bit integers; 50% of transactions are read-only.

3.4.2 Scalability

In Figure 3.2, we evaluate the scalability of RingSTM on a concurrent red-black tree. The benchmark uses random 20-bit values, with 50% lookup transactions and the remaining transactions evenly split between inserts and removals. The tree is pre-populated with $2^9$ unique elements. Not surprisingly, we find that all RingSTM variants scale well, and that performance is on par with our TL2 implementation until about 14 threads, at which point ring contention (competing to acquire ring entries) dampens the rate of scaling; we discuss this property further in Section 3.4.4.

Finding the right match of filter size to workload is important: at high thread levels, the 32-bit filters are too small, resulting in decreased scalability due to increased aborts from false positives. After 8 threads, even the 1024-bit filters cause false positives. Figure 3.3 contrasts the overhead of false conflicts with the cost of maintaining larger filters for the same RBTREE workload. Each bar represents the number of commits
Figure 3.3: Commit/Abort ratio for the red-black tree of Figure 3.2, with RingSTM using 32, 1024, and 8192-bit filters.

per aborted transaction, using a TL2-like algorithm as an approximation of the true commit/abort ratio. For smaller filter sizes, the lower ratio indicates that there are significantly more aborts; when total throughput remains on par with TL2 despite the depressed ratio, it is a consequence of the lower overhead of maintaining smaller filters. At 16 threads, even 8192-bit filters introduce too many false conflicts, resulting in a cross-over between RingSTM-8192 and TL2 in Figure 3.2.

Lastly, we observe that for high thread levels, RingSW performance is slightly lower than RingSTM, and that RingR performs best, though by only a small margin. These characteristics are directly related to write-back. At high thread levels, multiple writers are likely to reach their commit point simultaneously. In RingSW, one of these writers must block before acquiring a ring entry, limiting parallelism. Likewise, in RingSTM, a writer may delay after writeback in order to preserve the suffix property.
The slightly better scalability of RingR relative to RingSTM can be attributed to the cost of privatization safety. For our workloads, which write fewer than 50 different locations, the improvement afforded by RingR remains below 5% in all experiments presented in this chapter. Given the negligible privatization overhead for our workloads, we omit further discussion of RingR. Since the three ring variants are equally affected by filter size, with RingSTM scaling better than RingSW whenever there are multiple concurrent writers, we also omit further discussion of RingSW. The behavior of RingSTM is characteristic of all three algorithms, and since it does not require overhead for privatization, we feel that its performance is most interesting.

3.4.3 Commit Overhead

To highlight the impact of $O(R)$ validation overhead, we consider the Random-Graph benchmark from the RSTM suite [MSH+06a; Roc09]. The benchmark maintains a graph, where nodes are stored in a list and each node maintains its neighbors in an adjacency list. Transactions add or remove nodes with equal probability, with new nodes receiving four random neighbors. The benchmark does not admit parallelism, but its large read sets (potentially hundreds of elements) and moderately large write sets (tens of elements) tax STM systems that rely on ownership records. In Figure 3.4, we observe that RingSTM performs between 33% and 50% faster than TL2, since TL2 incurs $O(R)$ validation overhead and $O(W)$ locking overhead at commit time.

3.4.4 Bottlenecks

We lastly consider the impact of the single global ring on scalability. Figure 3.5 shows a hash table benchmark. The hash table stores 8-bit integers in 256 buckets with chaining, and insert, lookup, and remove operations are equally likely. The data
Figure 3.4: RandomGraph benchmark with 50% node inserts and 50% node removals.

Figure 3.5: Hash table benchmark with 256 buckets, 8 bit keys, and 33% lookups.
structure should scale almost linearly, but transactions are short enough that global shared variables (the TL2 timestamp and the ring) become bottlenecks.

As expected, there is a point at which coherence traffic on the global TL2 timestamp dominates, and the benchmark ceases to scale. Ring contention also becomes a bottleneck, with worse effect than in TL2 for several reasons. First, an update to the ring causes multiple cache misses in concurrent threads (the ring entry and the \texttt{ring.index} reside in separate lines, and write filters often reside in multiple lines). Secondly, the writer must update the ring entry after completing writeback; if concurrent transactions have already polled for conflicts, then the writer will take a write miss. Third, initializing a ring entry takes several cycles; during initialization, the ring is effectively locked. Lastly, as in TL2, attempts to increment a single global integer inherently serialize otherwise nonconflicting transactions.

3.5 Summary

RingSTM avoids the use of per-location metadata, instead using a global ring of Bloom filters that can each be evaluated in $O(1)$ time via a simple intersection. In its default configuration, RingSTM has no privatization problem, requires a single RMW operation per transaction, is inherently livelock-free, and has constant space overhead to maintain its read set. RingSTM offers an attractive alternative to orec-based STM algorithms for a number of workload characteristics. When privatization is frequent, or existing privatization mechanisms are too expensive even for limited use, RingSTM provides low-cost privatization. When transactions are used for thread-level speculation [BMT+07; vPCC07], RingSTM offers a shorter commit sequence than orec-based algorithms. In workloads with high ratios of read-only transactions, or workloads with
large read and write sets, we expect RingSTM to offer lower latency due to decreased metadata manipulation.

RingSTM is sensitive to its ring and filter sizes. As future work, we plan to develop a variant that adapts these parameters, as well as a “profiling mode”, in which additional bookkeeping permits the programmer to distinguish between real conflicts and conflicts due to filter imprecision; the programmer can then statically change the filter size or hash functions. We are also investigating nonblocking implementations, suitable for use in operating system interrupt handlers (which cannot tolerate blocking [RHP+07; RRP+07]).

Lastly, we have begun to look at hardware optimizations for RingSTM. Our hope is to identify primitives that are not specific to STM, but that can alleviate the cost of shared memory communication, further accelerating RingSTM without committing hardware manufacturers to a fixed TM design. We believe that the fixed-size metadata requirements of RingSTM will help keep RingSTM-specific hardware simple.
4 Reducing Memory Ordering Overheads

So far, we have looked at algorithmic issues that affect the latency of transactions. To evaluate the benefit of our proposals, a library-based STM implementation sufficed. However, for peak performance and programmability, compiler support is necessary. In addition to cloning functions called from transactional and nontransactional contexts, and automatically instrumenting the loads and stores of transactional regions, an optimizing transaction-aware compiler can identify and eliminate redundancy within the instrumentation of successive loads and stores.

Past work has focused on redundant metadata operations. In this chapter, we consider the specific case when transactions are run on processors with relaxed memory consistency models. In the pseudocode presented in previous chapters, we included explicit memory fences to order accesses to metadata and program data in this setting. Each fence typically introduces substantial latency, and so we now look at how the compiler can find and eliminate redundant fences, in a manner compatible with a wide variety of STM algorithms. Along the way, we will identify certain STM characteristics for which our optimizations eliminate both memory fences and validation instructions.
4.1 The Cost of Memory Fences

Attempts to characterize the sources of latency in STM implementations [DS07; MM08; MSH+06a; SATH+06] have typically ignored the cost of memory fences. This omission is perhaps because those STM systems have, coincidentally, been built for x86 or SPARC platforms, on which write-before-read (WBR) is the only ordering not guaranteed by the processor. Moreover atomic read-modify-write instructions, including CAS, implicitly incorporate a full memory fence on these machines, and such instructions are frequently used in places where a fence would otherwise be required.

Although STM research has assumed a relatively strict memory model, market penetration of CPUs with relaxed memory consistency may be at an all time high, due to the brisk sales of modern video game systems. These systems all employ processors based on IBM’s POWER architecture [Pau08], with the XBox360 providing three multithreaded POWER cores and the PlayStation 3 using a single POWER-based core with seven Cell SPE cores. On these consoles, and on high-end servers, the cost of excessive memory fences can be a significant source of latency if transactions are used for rendering [SSD+08], thread-level speculation [BMT+07; vPCC07], or library and systems code [RRP+07].

Figure 4.1 shows latency for a single-threaded execution of a red-black tree microbenchmark. A single coarse lock results in roughly 40% slowdown relative to unsynchronized code. Transactions (using a TL2-like STM algorithm [DSS06]) cause a 6× slowdown on the UltraSPARC III (again relative to unsynchronized code). If we (incorrectly) leave out fences on the POWER4, the slowdown for transactions is almost exactly the same. When we add the necessary fences, however, slowdown increases to 11×. Even if efforts to minimize the difference between coarse locks and transactions
Figure 4.1: Single thread latency of a common RBTree benchmark, normalized to an unsynchronized implementation, on UltraSPARC III and POWER4 processors. STM experiments used a TL2-like algorithm.

(the second and third bars of Figure 4.1) succeed, transactions may not be viable on CPUs with relaxed memory consistency due to the overhead of fences.

The fundamental problem causing the dramatic performance gap between correctly fenced STM code and unfenced code in Figure 4.1 is not that fences are needed, but that naïve transactional instrumentation based on a simple API results in more fences than are necessary for correctness. In this chapter, we present compiler optimizations to specifically target these fences. Our optimizations allow temporary read inconsistencies within transactional code, but do not introduce the need for sandboxing.

The simplest way to eliminate fences is to batch transactional reads within a basic block. This batching is beyond the ability of traditional redundancy elimination, as it requires knowledge of the underlying STM implementation to safely reorder accesses to volatile metadata while aggregating read-before-read (RBR) memory fences. We aug-
ment batching of reads with optimizations that hoist or delay metadata accesses beyond the boundaries of basic blocks, thereby removing fences in loop bodies and across complex control flows. Our techniques complement existing techniques for eliminating redundant transactional instrumentation [ATLM+06; HPST06; WCW+07; WMvP+09].

Our base set of optimizations is conservative and could be applied by an STM-aware compiler without violating isolation or introducing races between transactional and private use of any location. Programmer-supplied annotations may enable additional optimizations for certain applications and run-time systems (such as those that are not publication-safe [MBS+08a]), allowing us to eliminate memory fences for reads that cannot safely be reordered in the general case.

In Section 4.2 we discuss the circumstances that necessitate memory fences in various STM systems, focusing on runtimes that manage conflicts at the granularity of individual memory words. We then present safety criteria that the compiler must obey when removing memory fences in Section 4.3. Section 4.4 discusses safe and unsafe optimizations. In Section 4.5 we give performance results, based on hand-transformation of code according to our optimizations. Using the transformed code, we show reductions of up to 89% in fences, up to 20% in per-transaction latency.

4.2 The Problem of Memory Ordering for STM

STM systems that perform conflict detection and versioning at the granularity of individual addresses typically expose a simple interface with four functions [DSS06; SATH+06], as depicted in Table 4.1. These functions interact with volatile global metadata to ensure that the transaction remains atomic, isolated, and consistent throughout its execution. Each function requires ordering between its metadata accesses and its
accesses to program data. In the following subsections we outline the situations that force this explicit ordering.

### 4.2.1 Ordering Transactional Reads

The ordering requirements for read instrumentation depend on many STM internals. Accesses to metadata must be ordered relative to accesses to main memory in all cases.

**Ownership Records with Sandboxing or Indirection**

In systems like LibLTx [Enn06], McRT [SATH+06], and Bartok [HPST06], sandboxing is used to detect inconsistencies during transaction execution. A transactional read checks metadata before accessing a location, but not afterward, and a read-before-read fence is necessary to ensure that the location is not read until after the metadata is checked. Similarly, indirection-based nonblocking STM algorithms [HLMS03; MSH+06a; MSS05] require a metadata access that is ordered before the first read of an object.

**Ownership Records without Sandboxing**

In word-based STM without sandboxing [DSS06; FFR08; MM08], metadata is typically read before a location is accessed transactionally, to ensure the location is not locked, and again after the access to ensure that there was no intervening write. These
requirements introduce a need for two read-before-read fences. The first performs the same role as in systems that use sandboxing. The second orders the access to memory before the post-read test. Depending on the STM algorithm, the post-read test may entail checking either a single global variable [SMSS06] or a location-specific timestamp [DSS06; FFR08; RFF06; RFF07; WCW07].

No Ownership Records

In JudoSTM without sandboxing [OCS07], and in RingSTM [SMvP08], no ordering is required before accessing program data, but a read-before-read fence is required between the access of program data and a test of global metadata. This test may trigger a full validation of previously-read locations, which must also be ordered after the read of the program data. The validation can introduce additional ordering requirements, but in the common case the post-read test indicates that no additional validation—or memory fence—is required.

4.2.2 Ordering Acquisition Before Update

STM implementations may acquire ownership of to-be-written locations eagerly (upon first call to \texttt{TM\_WRITE} for each address) or lazily (at commit time). When eager acquire is used, the STM may choose to perform the write directly to main memory (direct update) and store the old value in an undo log, or to buffer the write (buffered update) in a redo log that is written to main memory during commit. With lazy acquire, writes must be buffered in a redo log.

With eager acquire and direct update, a fence is required to provide write-before-read/write ordering after an orec is acquired, to ensure that the orec is owned before the transaction modifies locations protected by the record. The fence also ensures that
any undo logging performed for the location uses a consistent value. For \( W \) writes to distinct locations, this results in \( W \) atomic RMW operations and \( W \) fences. On the x86 and SPARC, a \texttt{CAS} provides the memory fence implicitly. On POWER, an explicit fence is required after each atomic RMW (an \texttt{LL/SC} instruction pair). The last of these fences also orders all metadata acquisition before the final validation in the commit sequence.

In systems that use lazy acquire and buffered update, the \( W \) memory fences can be collapsed into a single fence. During execution, the transaction issues its writes to a private buffer without the need for any memory ordering. Then, at commit time, all \( W \) locations are acquired in a tight loop that issues \( W \) atomic RMW operations. Following the last of these, a single write-before-read/write fence orders acquisition before subsequent validation and writing. Similarly, when eager acquire is coupled with buffered update, only a single fence is necessary to ensure write-before-write ordering between all acquire operations and all write-backs, even though locations are acquired throughout the execution of the transaction (via \( W \) RMW operations). In JudoSTM and RingSTM, which use buffered update but no ownership records, there is only a single RMW operation at commit time, and a single fence.

Regardless of the acquisition protocol, an additional write-before-write fence is required during the commit sequence before releasing ownership, to order the last update to memory before the first ownership record release. On x86 and SPARC, this ordering is implicit; on POWER, it is provided through an LWSYNC. For eager acquire with direct update, the total memory fence overhead for \( W \) distinct writes is \( W + 1 \) fences. For buffered update (with either lazy or eager acquire), the total overhead is 2 fences. On relaxed machines such as POWER, buffered update thus avoids \( W \) fences. It also
allows the processor to reorder the instructions comprising write instrumentation; with
direct update, per-write fences preclude this.

### 4.2.3 Additional Ordering Constraints

Certain additional constraints can be ignored in the common case. When remote abort is possible, transactions must test their status regularly (often on every transactional read or write). To avoid allowing a transaction to acquire locks or abort others when it is aborted, it may be preferable to check this status early in the read or write sequence. However, it is correct to wait until the end of the sequence, in which case the previously described fences are sufficient. We assume that remote aborts are uncommon; when they are performed, additional ordering may be required.

When a transaction reads a location it has already written, some fences can be avoided: Under lazy acquire (with or without ownership records), all fences can be skipped if the value is found in the write set. Under eager acquire without sandboxing, the second test of metadata can usually be avoided. Lastly, if transactions read global metadata at begin time, it may be necessary to order such reads prior to the transaction’s execution. Similarly, when epoch-based memory reclamation is used in unmanaged languages [HSATH06], epoch updates must be ordered, via a constant number of memory fences during transaction begin and end.

### 4.3 Safety Criteria for Reducing Memory Fences

Typically, fence instructions are expensive, both because the fence causes a pipeline stall, and because the fence prevents the processor from exploiting potential instruction-level parallelism (an opportunity cost). Previous studies suggest that lazy acquire need
not reduce performance [DSS06; OCS07; SDMS09], is most compatible with the Java memory model [MBS08a], and avoids livelock. Since lazy acquire with buffered update also avoids the requirement for fences on transactional writes, for the remainder of this chapter we focus on reducing the fence requirements for transactional reads, and assume the underlying STM will use buffered updates.

The top half of Listing 4.1 depicts a naïve transformation of a program with two transactional reads for an orec-based STM with prevalidation and postvalidation, such as TL2 or tinySTM. A delay cost is experienced on lines 5, 7, 13, and 15. Quantifying opportunity cost is more subtle; one example is that the write set lookup on lines 9–10 is explicitly ordered after line 7, though it could be executed in parallel with the lookup on lines 1–2. In the worst case, this instrumentation will limit reordering in both the compiler and processor. The lower half of the listing depicts a reordering that halves the number of fences without compromising correctness. Additionally, this reordering increases the window in which the CPU can leverage out-of-order execution.

In this section we discuss the safety criteria that a compiler must obey when eliminating memory fences incurred during transactional instrumentation.

For our optimizations, we assume that the compiler converts high-level code to low-level instructions in four steps. First, the compiler performs traditional analysis and optimization, such as pointer analysis, redundant read elimination, silent store elimination, and register promotion. Second, the compiler instruments the begin and end sequence of the transactional block, to ensure that metadata is initialized at transaction begin, and that transactional state is committed on all possible exit paths from the transactional context. For functions called from both transactional and nontransactional contexts, this step may require that the functions be cloned. Third, the compiler inserts checkpoint instructions to preserve the state of private, non-local variables that may be
a = TM_READ(X)
b = TM_READ(Y)

Naïve Transformation:

1 if (is_in_write_set(&X))
2   a = value_from_redo_log(&X)
3 else
4   prevalidate(&X)
5   read-read-fence
6   a = X
7   read-read-fence
8   postvalidate(&X)
9 if (is_in_write_set(&Y))
10  b = value_from_redo_log(&Y)
11 else
12  prevalidate(&Y)
13  read-read-fence
14  b = Y
15  read-read-fence
16  postvalidate(&Y)

Optimized Transformation:

1  prevalidate(&X)
2  prevalidate(&Y)
3  read-read-fence
4 if (is_in_write_set(&X))
5   a = value_from_redo_log(&X)
6 else
7   a = X
8 if (is_in_write_set(&Y))
9   b = value_from_redo_log(&Y)
10 else
11  b = Y
12  read-read-fence
13  postvalidate(&X)

Listing 4.1: Naïve and optimized transformation of two transactional reads, using a TL2-like runtime.

modified by a transaction that aborts. Lastly, the compiler replaces heap accesses (loads and stores) within a transaction with their instrumented equivalents.
4.3.1 Decomposing Read Instrumentation

In the naïve transformation of Listing 4.1, each heap access is replaced with a sequence of instructions that correspond to a transactional read. We describe the read as occurring in three phases. The prevalidation phase (line 4) ensures that the address to be accessed is not currently locked. The dereference phase (lines 1, 2, and 6) searches for a speculative write to the location, and if none is found, reads from main memory. The postvalidation phase (line 8) ensures that the address remained constant during the read step; that is, postvalidation ensures that a concurrent write did not take place between the prevalidation and read steps. There is no reason why these three phases must occur within a single API call. By decomposing the read instrumentation into three separate API calls, corresponding to these three phases, we can create opportunities for the compiler to reorder instrumentation to minimize the number of memory fences.

For a single transactional read, order must be preserved between the prevalidation step and the dereference within the read step, and between the dereference and the postvalidation step. However, for multiple transactional reads to independent locations, there is no required order among the reads themselves. If there is no data dependence between two reads, then it is safe for the compiler to reorder the corresponding sub-steps of the corresponding transactional reads. For certain weak semantics, this criterion may not suffice due to potential races with nontransactional code [MBS+08a]. We will discuss semantics in greater detail in Chapter 7. Throughout this chapter we assume a strong semantics that precludes such races, such as annotation-based semantics [SDMS08].
windows between the memory fences would be larger. Note, however, that an ideal memory fence reduction may lead to increased windows for inter-transaction conflict (we will discuss this in Section 4.3.3), and that it may introduce additional fences in workloads with frequent read-after-write accesses, where the tests on lines 1 and 9 are expected to succeed. Finally, we must impose several limits on the use of read values prior to postvalidation.

### 4.3.2 Safety Criteria for Postvalidation Delay

Reads that are candidates for fence-reduction optimizations may appear in separate basic blocks. We outline criteria for reducing fences below. We begin with the following two invariants, where “precede” implies the existence of a read-before-read memory fence between the corresponding instructions:

- Prevalidation of location X must precede the dereference of X.
- Postvalidation of location X must precede any potentially unsafe use of the value read from X.

Our optimizations do not explicitly reorder reads of program data, but may allow the compiler to do so when profitable. Regarding the above invariants, the first is straightforward. For the second, there is a potential unsafe window after the value is read and before postvalidation. To determine how far one can safely delay a postvalidation, we begin with the following seed:

A value that has been read but not postvalidated is considered unsafe.

---

2On the POWER architecture, this ordering may be provided with either the instruction-sync (ISYNC) or lightweight-sync (LWSYNC), depending on the surrounding STM library code.
We conservatively require that on any criterion that terminates postvalidation delay, all deferred postvalidation is performed. There are six categories of operations that require analysis:

“+” (arithmetic operation that causes no fault): This operation does not generate a fault, but propagates unsafeness, i.e., if any of its operands is unsafe, the result is unsafe. Divide may be placed in this category if the compiler inserts a dynamic check for a non-zero divisor, and if the check fails, inserts a postvalidation before the division. We include nonfaulting type casts (e.g., C++ reinterpret and const casts) in this category: these casts do not generate a fault, but propagate unsafeness to their result. Comparison operations also fall into this category, with branches addressed below.

“*” (address dereferencing): Dereferencing may generate a segmentation fault or bus error. Thus none of its operands can be unsafe. This is a leaf condition to terminate a postvalidation delay.

“=” (assignment): When the right hand side of an assignment is unsafe, this operation does not generate a fault, but propagates unsafeness through memory. In particular, any value (possibly) read from the store address prior to the next postvalidation must also be treated as unsafe. Note that this condition requires that a read after write may require postvalidation.

If the assignment uses transactional write instrumentation, then the store address (the left hand side) may be unsafe, since the address will not be used in a true assignment until after the transaction validates at commit time. However, if the assignment is not performed via transactional instrumentation (for either a heap or stack variable), then if the address is unsafe, postvalidation is required before the assignment. Failure
to perform postvalidation may expose speculative reads to concurrent, nontransactional threads if the store address is visible to concurrent threads. This criterion may be softened for uninstrumented stores to provably thread-local locations, such as those that have been checkpointed by the compiler, or those that will be deallocated on abort.

“branch / jump” (control flow): Use of an unsafe condition in a conditional jump may be dangerous as it may lead to erroneous execution and infinite loops. To ensure safety, a postvalidation can be placed before an unsafe conditional jump or at the beginnings of the jump target and the fall through path. The latter alternative exposes opportunities for further postvalidation batching. Additionally, the jump target must not be unsafe (e.g., the target of an unsafe function pointers). Allowing otherwise could permit jumps to arbitrary transactional or nontransactional code, which could then cause externally visible effects. (For example, consider branching to an unsafe address that stores the instruction \texttt{mov [r1], r2}. The instruction may be in a block that assumes \texttt{r1} to hold the address of a stack variable, but the unsafe jump may follow an instruction that sets \texttt{r1} to a value that looks like a global address.) Using these criteria, back-edges taken due to nonspeculative conditions (e.g., most \texttt{for} loops) will not terminate postvalidation delay.

“transaction end” (control flow): Some STM runtimes do not perform a final validation of read-only transactions. If the compiler is not certain that a transaction will perform at least one speculative write (which forces the transaction to validate all reads at commit time), then no values may be unsafe at the point where the transaction attempts to commit. This condition may be provided implicitly by the underlying STM, if it always validates before committing a read-only transaction with un-postvalidated reads.
“function call” (control flow): When the side effects of a function call are unknown, postvalidation cannot be delayed beyond the function call. When the function is known not to use unsafe values, then postvalidation can be delayed until after the function call.

4.3.3 Performance Concerns

While the above safety criteria appear sufficient to prevent incorrect behavior, analysis and optimization using these criteria may lead to unexpected performance degradation, which may necessitate some tuning by the programmer. The following tradeoffs preclude any static notion of optimal fence reduction.

Instrumentation and Function Call Overhead: In STM systems that use ownership records, the set of un-postvalidated addresses must be tracked dynamically. Furthermore, the corresponding pre- and postvalidation code may be too large for consideration for inlining by the compiler. When fences are removed for a set of $K$ reads, up to two additional function calls may be required.

Early False Conflicts: Let us suppose that a loop performs $K$ reads, and that prevalidation of those $K$ reads can be hoisted out of the loop. At the point where prevalidation completes, the transaction has effectively added all $K$ locations to its read set, and a concurrent write to any of those $K$ locations by another transaction will force the reader to abort during postvalidation. If a write to the $K$th location occurs before the reader’s $K$th read, then performing early prevalidation of the $K$th location prevents a valid schedule in which the writer commits during the reader’s loop, but prior to the read of $K$. Thus early prevalidation can prevent transactions from succeeding.\footnote{With TL2-style timestamps, the successful schedule is explicitly forbidden regardless of the timing of prevalidation.}
RingSTM, where the postvalidation of one read is effectively a prevalidation of the next read, deferred postvalidation causes the same unnecessary abort.

**Delayed Abort Detection:** Conversely, in the same loop performing $K$ instrumented reads, it may happen that after the first read, to location $L$, a concurrent writer commits a change to $L$. If the reader defers postvalidation of $L$ until after all $K$ reads, then the subsequent $K - 1$ reads are all unnecessary; earlier postvalidation could have identified the conflict, and enabled the reader to restart earlier. Thus postvalidation delay can prolong the execution of a doomed transaction.

### 4.3.4 Implementation Challenges

The above safety criteria form the foundation of a compiler algorithm to eliminate memory fences by moving the pre- and postvalidation calls that must accompany any transactional heap access. However, several challenges remain, which favor the use of correct approximations of a general algorithm, as discussed in the following section. The main challenges to implementing a general algorithm are:

**Pointer analysis precision:** The precision of the compiler’s pointer analysis determines the degree to which the compiler can ensure that an unsafe address is not aliased.

**Logging overhead:** In STM algorithms that use ownership records, such as TL2 and tinySTM, the individual locations that have not been pre- or postvalidated must be logged. This complicates the process of delaying validation across complex control flows.
STM specificity: In addition to not requiring per-location postvalidation, the JudoSTM and RingSTM algorithms do not require any form of prevalidation. A compiler targeted to these specific runtimes can elide much analysis, logging, and instrumentation. Additionally, in these systems postvalidation delay serves to reduce both the instruction count and the memory fence count, since the common-case instruction count for postvalidation is not input-dependent.

4.4 Eliminating Redundant Memory Fences

We now consider five techniques to reduce memory fences while obeying the above safety criteria. These techniques are largely independent, but all benefit from a transaction-aware partial redundancy elimination. This analysis serves two roles. First, it removes provably redundant transactional heap accesses, that is, multiple transactional reads to the same variable without an intervening write. Second, when a transactional read is performed on all paths of a flow graph, the analysis hoists that read to the root of the graph.

4.4.1 Removing Fences Within a Basic Block

The first, and most straightforward, mechanism to reduce memory fences analyzes individual basic blocks. For a single block with transactional reads to locations $X$ and $Y$, if both addresses are known at the entry of the basic block, then the reads may both be hoisted to the top of the block, and then transformed via the simple transformation in Listing 4.1. In practice, this technique typically permits multiple fields of a single object to be read in a single batch.
More generally, for a set of addresses \( \{A_1 \ldots A_n\} \), where all addresses in the set are known at the start of a basic block, all metadata accesses that must be issued before the locations are read (the `prevalidate()` calls in the top half Listing 4.1) can be hoisted to the beginning of the block and combined. All subsequent dereferences (e.g., lines 1, 2, and 6 in the top half of Listing 4.1) can be moved to directly above the first use of any of the results of those reads, and all metadata accesses that must be issued after locations are read (the `postvalidate()` calls) can be combined and moved to as late as immediately before the first use of any of the results of the reads.

For STM systems that do not require prevalidation, such as RingSTM and Ju-doSTM, the transformation is slightly simpler. The transformation differs from the bottom section of Listing 4.1 in that lines 1–3 can be removed, and lines 13–14 can be replaced with a single `postvalidate()` call. A simple augmentation of def-use analysis is sufficient to place the calls that validate transactional reads: the result of a transactional read cannot be used unless there is a read-before-read fence and then a call to `postvalidate()` between the transactional read and its first use.

### 4.4.2 Tight Loop Optimizations

Listing 4.2 depicts a simple loop in which many locations are read from a single array. Since there is only one read per iteration, our previous optimization is unable to eliminate memory fences. While loop unrolling can increase the opportunity for fence reduction, a simpler alternative exists: all prevalidation can be performed prior to the first memory access, and all postvalidation can be delayed until after the last memory access. In this manner, \( 2n \) memory fences can be reduced to 2.

The orec code is noticeably more complex: since each location must be prevalidated and postvalidated individually, the loop must be replicated for each phase of the
Listing 4.2: Eliminating fences for reads issued within a loop.

heap access (prevalidation, dereference, postvalidation). For loop bodies that contain conditional reads, writes, or function calls, the transformation is much more difficult with orecs, and may also result in substantially more function call overhead, depending on whether the individual prevalidations and postvalidations can be inlined.
4.4.3 Removing Final Postvalidation

When a writing transaction commits, it must follow a protocol in which all locations are acquired and then all reads are validated, to ensure isolation. While the read-set validation may have an \(O(1)\) fast path, the general requirement that every writer transaction ensures the validity of its entire read set during its commit phase allows further optimization of the code in Listing 4.2. Specifically, since the reads in the loop are used only for safe arithmetic, and then for an instrumented write, no postvalidation is required (lines 9–11 of the middle section of the listing, lines 6–7 of the bottom section).

As a simple approximation of this optimization, the compiler may eliminate postvalidation when the calling transaction performs at least one write, and all code paths from the postvalidation to the transaction commit point do not contain instrumented reads. For maximum effect, we also clone functions that can be called as the last action of a writing transaction, and perform this optimization within those function bodies. This simulates the effect of aggressive inlining or whole-program analysis. This optimization typically eliminates only one memory fence. However, it may lead to a noticeable reduction in instructions for orec-based STM, since a batch of \(n\) postvalidation operations may be avoided.

4.4.4 Dynamically Checked, Unsafe Use

Our safety criteria typically require that fault-generating operations not use operands that are the result of un-postvalidated transactional reads. However, as discussed in the case of division, a fault-generating operation can be made safe by inserting a dynamic check (in this case, test for zero). When the test fails, a postvalidation is required be-
\[ q = x->f1[x->f2] \]

Becomes:

```c
1   if (is_in_write_set(&x->f2))
2       t = value_from_redo_log(&x->f2)
3   else
4       t = x->f2
5       read-read-fence
6       postvalidate()
7   if (is_in_write_set(&x->f1[t]))
8       q = value_from_redo_log(&x->f1[t])
9   else
10      q = x->f1[t]
11      read-read-fence
12      postvalidate()
```

Or, optimized:

```c
1   if (is_in_write_set(&x->f2))
2       t = value_from_redo_log(&x->f2)
3   else
4       t = x->f2
5   if ((t < 0) || (t >= MAX))
6       read-read-fence
7       postvalidate()
8   if (is_in_write_set(&x->f1[t]))
9       q = value_from_redo_log(&x->f1[t])
10  else
11     q = x->f1[t]
12    read-read-fence
13    postvalidate()
```

**Listing 4.3:** Dynamically checked, un-fenced array indexing when array bounds are known.

fore performing the division to distinguish between failed speculation (i.e., transaction conflicts) and program bugs.

Similarly, dereferences may be dynamically sterilized without requiring postvalidation. Listing 4.3 shows one example. A transactional read of field \( x->f2 \) de-
terminates an array index. If the array size is statically known (represented by \( \text{MAX} \)),
then the index may be used without postvalidation, so long as it is within the range
\( (0 \ldots \text{MAX} - 1) \). This optimization permits the read of the index to be batched with
the read of the array at that index, with a possible postvalidation only if the index is
out of range. For STM systems that use ownership records, this analysis requires that
the compiler know the granularity of the location to ownership record mapping, so that
calls to \text{prevalidate()} can be made with the correct parameters.

The bottom section of Listing 4.3 depicts the transformation for RingSTM. By
inserting a dynamic test on lines 5–7, we can guarantee that any fault generated by the
read to \( x->f1[t] \) is due to a program bug, not due to unsafe optimization causing an
out-of-thin-air read. Since there is no longer a risk of a memory fault, the value read
transactionally from \( x->f2 \) can be used unsafely on lines 8, 9, and 11. The validation
call on line 13 is ordered after both transactional reads, and before any use of variable
\( q \), and ensures that the transaction aborts if either of the reads was inconsistent.

### 4.4.5 Speculative Read Hoisting

The last optimization we consider is the most risky: either the programmer or the
compiler can hoist reads taken on one path of a branch to above the branch, in order to
increase the potential for fence reduction. This optimization increases the read set size
by reading extra locations; consequently it can create false conflicts with concurrent
writers and can introduce additional validation instructions at commit time. However,
by hoisting reads even when they are not taken on all paths, the compiler can avoid
fences in hot loops, such as those for data structure traversal. Furthermore, as in the
code of Listing 4.4, the compiler may limit the use of this optimization to reads of fields
of a single object.
1 bool find(goal)
2 node x = TM_READ(tree_root)
3 while (x)
4     v = TM_READ(x->val)
5     if (v == goal)
6         return true
7     x = (v < goal) ? TM_READ(x->r) : TM_READ(x->l)
9 return false

Becomes:

1 bool find(goal)
2 node x = TM_READ(tree_root)
3 while (x)
4     v = TM_READ(x->val)
5     t1 = TM_READ(x->r)
6     t2 = TM_READ(x->l)
7     if (v == goal)
8         return true
9     x = (v < goal) ? t1 : t2
10 return false

Listing 4.4: Hoisting a transactional read. This transformation can cause incorrect behavior.

Listing 4.4 depicts a binary search tree lookup. The transformed code hoists two conditional reads, so that they can be batched with the read of the node’s key. Without the optimization, the loop is not a candidate for fence reduction; with the optimization, the fence count will be halved on every loop iteration. For tree and list operations, this optimization can dramatically reduce the dynamic fence count.

Unfortunately, this optimization may cause erroneous behavior due to either publication [MBS+08a] or explicit violations of type safety. Publication-related errors are not an issue when the application does not use a flag-based publication idiom, or when the underlying STM is publication-safe. Type safety-related errors are more nuanced. Suppose that variable \( x \) in Listing 4.4 is of type \( \text{Foo} \), and the programmer explicitly
casts an address to type \texttt{Foo}. Let us further suppose that it is never correct for a \texttt{Foo} to have the field \texttt{val} equal 7, but that the programmer is using that value to indicate that field \texttt{l} is not on the same page as \texttt{r}, and \texttt{x->l} is not even a valid address. In this case, executing line 6 of the optimized code will result in a segmentation fault and incorrect program termination. While it is possible to provide a dynamic check and only batch reads that reside on the same page, we expect that there are many more opportunities for hoisting to lead to errors in programs that violate type safety, even if the underlying STM is publication-safe. Consequently, we expect speculative hoisting to serve as a programmer tool, rather than an optimization explicitly performed by the compiler, even if the compiler can prove that type safety is not violated.

### 4.5 Experimental Evaluation

We now analyze the impact of memory fence reduction through a series of experiments using the STAMP benchmarks [CCKO08]. We conducted all tests on an IBM pSeries 690 (Regatta) multiprocessor with 16 dual-core 1.3 GHz POWER4 processors running AIX 5.1. All benchmarks and STM runtime libraries were written in C and compiled with gcc v4.2.4. Each data point is the average of ten trials.

Experiments labeled “Orec” use a lazy acquire, buffered update, timestamp-based STM patterned after the per-stripe variant of TL2 [DSS06]. Orec uses an array of 1M ownership records, and resolves conflicts using a simple, blocking contention management policy (self-abort on conflict). “Ring” experiments use the single writer variant of RingSTM [SMvP08] (results are similar for the other variants). Ring uses 8192-bit filters, a single hash function, and 32-bit summary filters. In both Orec and Ring, fast-path validation issues a memory fence, tests a global, and continues.
Both Orec and Ring serialize writing transactions on a single global variable (a
global timestamp and a ring head pointer, respectively). Our optimizations reduce the
latency of individual transactions, but do nothing to avoid these inherent bottlenecks.
In separate experiments using a sandboxed runtime, we determined that for the bench-
marks presented in this chapter, the point of serialization is not usually a bottleneck.

4.5.1 Optimization Levels

In our evaluation of the STAMP benchmarks, we compare six levels of optimization.
The optimizations were achieved through hand-instrumentation of the STAMP code,
but could be performed automatically by the compiler. The “Base” code uses STAMP
version 0.9.9, modified only to support our basic STM API. The “Base+” optimizations
manually inline some red-black tree helper functions, and eliminate redundant reads in
the red-black tree and linked list. These optimizations increased the scope of later
optimization levels. “Batched” code eliminates memory fences within basic blocks,
and “TLFP” adds “Tight Loop” and “Final Postvalidation” optimizations to the Batched
optimizations. “SRH” adds “Speculative Read Hoisting” to TLFP, most notably to the
red-black tree and linked list. “NoFences” uses an STM runtime with no memory
fences on reads. This is not a correct STM implementation, but provides a lower bound
for single-thread speedup. The STAMP benchmarks do not provide an opportunity to
evaluate the “Dynamically Checked, Unsafe Use” optimization.

We also generated a custom version of the benchmarks (“Ideal”) that aggressively
hand-optimizes postvalidation fences through an analysis similar to def-use. Ideal pro-
vides all of the benefits of SRH, but without introducing unnecessary reads or increas-
ing the conflict window. Since this analysis does not consider prevalidation, we applied
**Table 4.2:** Total prevalidation fences for STAMP benchmarks. These fences are incurred only by the Orec runtime.

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Base+</th>
<th>Batched</th>
<th>TLFP</th>
<th>SRH</th>
<th>Ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>genome</td>
<td>93.86 M</td>
<td>93.74 M</td>
<td>93.71 M</td>
<td>93.71 M</td>
<td>48.73 M</td>
<td></td>
</tr>
<tr>
<td>intruder</td>
<td>164.21 M</td>
<td>163.76 M</td>
<td>155.57 M</td>
<td>155.57 M</td>
<td>99.44 M</td>
<td></td>
</tr>
<tr>
<td>KMeans_high</td>
<td>20.33 M</td>
<td>20.33 M</td>
<td>20.33 M</td>
<td>1.07 M</td>
<td>1.07 M</td>
<td></td>
</tr>
<tr>
<td>KMeans_low</td>
<td>16.6 M</td>
<td>16.6 M</td>
<td>16.6 M</td>
<td>0.87 M</td>
<td>0.87 M</td>
<td></td>
</tr>
<tr>
<td>labyrinth</td>
<td>5.65 K</td>
<td>5.65 K</td>
<td>5.39 K</td>
<td>5.39 K</td>
<td>5.39 K</td>
<td></td>
</tr>
<tr>
<td>vacation_high</td>
<td>373.9 M</td>
<td>373.71 M</td>
<td>373.53 M</td>
<td>373.53 M</td>
<td>212.25 M</td>
<td></td>
</tr>
<tr>
<td>vacation_low</td>
<td>256.45 M</td>
<td>256.41 M</td>
<td>256.38 M</td>
<td>256.38 M</td>
<td>144.66 M</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.3:** Total postvalidation fences for STAMP benchmarks. These fences are incurred by the Orec and RingSTM runtimes, but the Ideal optimization level does not apply to the Orec runtime.

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Base+</th>
<th>Batched</th>
<th>TLFP</th>
<th>SRH</th>
<th>Ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>genome</td>
<td>92.65 M</td>
<td>92.65 M</td>
<td>92.65 M</td>
<td>47.67 M</td>
<td>47.62 M</td>
<td></td>
</tr>
<tr>
<td>intruder</td>
<td>153.63 M</td>
<td>153.63 M</td>
<td>153.63 M</td>
<td>97.45 M</td>
<td>96.94 M</td>
<td></td>
</tr>
<tr>
<td>KMeans_high</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>KMeans_low</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>labyrinth</td>
<td>5.33 K</td>
<td>5.33 K</td>
<td>5.33 K</td>
<td>682</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vacation_high</td>
<td>373.53 M</td>
<td>373.53 M</td>
<td>373.53 M</td>
<td>211.34 M</td>
<td>202.39 M</td>
<td></td>
</tr>
<tr>
<td>vacation_low</td>
<td>256.38 M</td>
<td>256.38 M</td>
<td>256.38 M</td>
<td>144.38 M</td>
<td>138.41 M</td>
<td></td>
</tr>
</tbody>
</table>

**4.5.2 Analysis**

In a single execution of the Genome benchmark (Figure 4.2), there are approximately 93M instrumented reads. The benchmark makes extensive use of a linked list, and thus speculative read hoisting (SRH) is very profitable, as half of the fences in an $O(n)$ list traversal can be removed, at the expense of at most one extra location read. The net result is that 45M fences are avoided with the RingSTM runtime, and 90M fences with the Orec runtime.
Figure 4.2: STAMP Genome benchmark.
Due to the size of read and write sets, our RingSTM configuration (with 8192-bit filters) does not scale well, though the single-thread throughput is 25% faster than Orec. Since batching eliminates less than 2% of the total fences, we observe little performance difference until the SRH optimization level. At this level, SRH performs within 10% of the NoFence curve, and our Ideal instrumentation is within 5%.

By increasing the number of locations read during tree traversal, SRH creates artificial conflicts among transactions, resulting in an 8% increase in aborts at 32 threads for RingSTM. Since the Orec runtime uses 1M ownership records, conflicts are detected at a much higher granularity, and extra tree accesses do not noticeably affect aborts. However, the benchmark still runs out of parallelism, at which point the optimizations do not offer much improvement.

The Vacation benchmark (Figure 4.3) makes extensive use of a red-black tree. There are 260M reads in a single execution of the benchmark, of which only 86K can be combined in the Batched level, for a savings of 32K fences in RingSTM. Since tree traversal is a hot code path, speculative read hoisting is very profitable. While the hoisting increases reads from 260M to 376M, it does so while almost halving the total memory fence count. In RingSTM, the count drops from 256M to 144M (138M for the Ideal curve). Orec experiences similar drops in both prevalidation and postvalidation. At one thread, these fence reductions result in a 31% improvement for the Ideal curve, but the execution is still 11% slower than if no fences were required.

Since there is significantly more unnecessary reading due to speculative read hoisting in the tree code, SRH experiences an 8% increase in aborts for the Orec runtime, and a 25% increase for RingSTM at 32 threads. Since Ideal achieves the fence reduction effects of SRH without the extra reads, there is no increase in aborts. This difference accounts for the slowdown for SRH at high thread counts. Again the bench-
Figure 4.3: STAMP Vacation (low contention). Results are similar for higher-contention workloads.
Figure 4.4: STAMP Intruder, RingSTM runtime. Trends are similar for the Orec runtime.

mark runs out of parallelism at 16 threads, at which point the optimizations cease to decrease latency.

Intruder (Figure 4.4) continues to demonstrate the above trends, with SRH and Ideal eliminating more than half of the performance lost due to memory fences. We note, however, that Intruder has many small transactions that serialize on global shared variables within the STM runtime. This leads to a slowdown beyond 8 threads. While fence reduction decreases latency for individual transactions, and thus continues to improve throughput even beyond peak scalability, it cannot inject scalability into a workload once the underlying STM algorithm encounters an inherent bottleneck.

The KMeans experiment in Figure 4.5 showcases the value of the Tight Loop and Final Postvalidation optimizations. Of the 20M instrumented reads performed by a single thread, the overwhelming majority are performed in a single loop. After tight loop optimizations, the sole remaining postvalidation call is the final instruction of
Figure 4.5: STAMP KMeans (high contention), RingSTM runtime. Trends are similar for the Orec runtime, and for higher contention workloads.

A writing transaction, and is thus eligible for removal. The other transactions in the code are also eligible for the Final Postvalidation optimization, resulting in an elimination of all fences in the RingSTM code, and all postvalidation calls in the Orec code (note, however, that prevalidation is still required for the Orec code, resulting in 1.07M fences total). The net result is a 29% improvement over the unoptimized code at 32 threads. Since our optimizations safely remove most validation instructions for RingSTM, single-thread performance is slightly better than NoFence. With Orec (not shown), single-thread performance is 2% slower for TLFP than NoFence, due to fences during prevalidation.

Labyrinth (Figure 4.6) exhibits surprising behavior: Tight Loop optimizations enable reordering of all but 1% of the reads in the benchmark, and eliminate 88% of the memory fences. The main loop being optimized, however, is also a contention hotspot. During a phase of execution, up to 400 loop iterations are performed by concurrent
threads, with each thread attempting to make the same update to an array. When any thread commits its update transaction, all concurrent threads should abort. However, when all postvalidation is removed from the loop, doomed transactions may run for hundreds of iterations before reaching their next postvalidation point.

Our RingSTM runtime affords the ability to constrain the number of outstanding postvalidations simply by counting the number of unsafe reads. Since postvalidation is a global event in RingSTM, once the count reaches some threshold, a single call can reset the count and ensure that all prior reads remain safe. In Figure 4.7, we vary this threshold from its default (\(\infty\)) down to 2. For modest values, we observe that the pathological behavior is broken.

Outside of tight loops, there are few opportunities to delay postvalidation of more than 4 reads within STAMP. Since there are diminishing returns as the batch size

Figure 4.6: STAMP Labyrinth, RingSTM runtime. Results are similar for the Orec runtime.
Figure 4.7: Varying frequency of postvalidation improves Labyrinth performance.

increases, it appears that 4 outstanding postvalidations may be a reasonable point for
the runtime to intervene and perform a postvalidation.

4.6 Related Work

Prior research into compiler optimizations for transactional memory focused on
systems with eager acquisition of locations and direct update [ATLM+06; HPST06;
WCW+07]. These efforts exposed redundant instrumentation through a decomposed
API, but only considered the x86 memory model. Thus for locations $L_1$ and $L_2$, if
the compiler could determine statically that both locations hashed to the same orec,
then prevalidation of the second location would be eliminated. Our work complements
this technique by identifying and eliminating memory fence redundancy when $L_1$ and
\textit{L2} hash to different orecs, and would be useful for eager orecc-based STM on relaxed memory models.

The STM algorithm used by Wang \textit{et al.} also required postvalidation [WCW+07], but did not consider batching postvalidation operations, since there is no redundancy within postvalidation instrumentation (apart from memory fences) for orecc-based STM. In non-orecc systems, where postvalidation typically polls a global variable and immediately returns, our technique eliminates both fences and excess polling instructions, and thus is profitable even on less relaxed memory models.

Most research into synchronization elimination focuses on atomic operations. However, von Praun \textit{et al.} identified techniques to eliminate memory fences, albeit those that accompany atomic operations [vPCCR06]. Our optimizations extend fence-elimination to STM algorithms, where the majority of fences do not accompany atomic operations. In STM, prevalidation behaves like \texttt{sync\_acquire}, and postvalidation serves as a \texttt{sync\_release}. However, the criteria for identifying redundant fences differ slightly: in STM, only the fence accompanying the last acquire is required (as opposed to the first), and the fences accompanying an acquire do not provide the necessary ordering for subsequent releases: For STM, only a prior release can provide the needed ordering.

\subsection{4.7 Conclusions}

In this chapter, we analyzed the specific case of STM running on processors with relaxed memory consistency models. Our evaluation showed that fences introduce substantial overhead, but that many of these fences can be eliminated by a transaction-aware compiler. We proposed a set of safety criteria that must be preserved by any optimization that reduces memory fences, and then proposed a number of different op-
timizations compatible with these constraints. In our experiments, safe optimizations, amenable to automatic compiler implementation, yielded improvements of more than 20% in some cases, while reducing fence counts by up to 100% in one workload.

The impact of prevalidation is perhaps the most remarkable: when a runtime uses oreccs and requires prevalidation, our optimizations are severely limited. The largest benefit was enabled by explicit source-level hoisting of reads, which served to co-locate reads within a basic block that could not safely be moved together without programmer knowledge. For runtimes that do not require prevalidation, such as RingSTM, a hand-approximation of whole program analysis was able to provide the same increase in throughput, without increasing aborts or requiring source-level hoisting.
The RingSTM algorithm of Chapter 3 was livelock-free, but in Chapter 4, we saw that its limited conflict detection granularity could limit its ability to scale in workloads with very large but disjoint write sets. Furthermore, while RingSTM could be augmented to provide starvation avoidance, it came at the cost of blocking all low priority writers. When memory fences are not an issue, when nonblocking progress is needed, when write sets are very large, or when hardware is available to accelerate STM performance, a design based on ownership records may be preferable to RingSTM. However, STM implementations with ownership records often admit livelock and starvation.

Contention management refers to the mechanisms used to ensure forward progress—to avoid livelock and starvation, and to promote throughput and fairness. Unfortunately, most past approaches to contention management were designed for obstruction-free STM frameworks, and impose significant constant-time overheads. Priority-based approaches in particular typically require that reads be visible to all transactions, an expensive property that is not easy to support in most STM systems. In this chapter we show how an STM implementation can provide good throughput and fairness under high contention, through mechanisms that incur low overhead when contention is low.
5.1 Contention Management in Software Transactional Memory

Orec-based STM algorithms typically achieve atomicity and isolation by acquiring written locations, either at encounter time (“eager acquire”) or at commit time (“lazy acquire”). The choice between these strategies can have a profound impact on latency and throughput, with eager systems vulnerable to livelock, and lazy systems suffering from higher latency on individual reads and at commit time.

Contention Management [GHP05b; SS05a] is the study of mechanisms to ensure throughput in STM. Originally proposed as an out-of-band mechanism to avoid livelock in obstruction-free STM implementations [HLMS03], contention management has grown into an eclectic set of heuristic policies, few of which are optimal in any provable sense, and many of which are specific to particular TM systems. The more sophisticated policies (e.g., those of Scherer and Scott [SS05a]) require extra bookkeeping on every memory access within a transaction.

No consensus has yet emerged on which forms of contention management might be best, or even on how to choose. In a broad sense, certain basic choices in STM design (e.g., use lazy acquire) might be considered contention management. Certainly admission control (e.g., serialize with a global lock if livelock is suspected) is a form of contention management. Most work, however, has focused on the narrower problem of conflict resolution, which chooses, when transactions conflict, which will continue and which will wait or abort. Popular current policies for conflict resolution include the following.

**Aggressive:** When an in-flight transaction $I$ wishes to access a location $L$ owned by transaction $W$, $I$ may simply abort $W$. While this policy ensures nonblocking
progress [HLMS03], it appears most prone to livelock [BMV+07] for workloads with even a modest amount of contention.

**Passive:** In TL2 [DSS06] and tinySTM [FFR08], transactions self-abort if they detect a conflict with a concurrent writer. This policy ensures good throughput in workloads with a regular access pattern.

**Polite:** In the original DSTM [HLMS03], a transaction that discovers a conflict defers to its competitor using bounded randomized exponential backoff. This allows many conflicts to resolve themselves without aborts. (If the backoff limit is exceeded, a waiting transaction aborts its competitor.)

**Karma:** In an attempt to favor the transaction in which the most work has been invested, Scherer and Scott track the number of transactional objects accessed by each in-flight transaction (across all attempts), and give priority to the one with the larger count [SS05a]. The “Polka” refinement of Karma adds an exponential backoff component to the decision of when to abort.

**Greedy:** By using visible reads (similar to reader/writer locks), and favoring transactions with an earlier start time, Guerraoui et al. are able, provably, to avoid both livelock and starvation, at the cost of high latency on every read even in contention-free workloads [GHP05b].

Though all these existing policies are effective in many situations, we are not aware of any that achieves the dual goals of (1) low overhead for low-contention workloads and (2) good throughput and fairness under high contention. In this chapter, we show that both goals can be achieved at the same time. We also accommodate user-defined priorities, something that matters a great deal in application domains like soft real time [GC08], but that is usually not achieved in existing TM systems. Even recent
single-CAS STM systems like RingSTM and JudoSTM, which are livelock-free, address starvation using mechanisms that block all low-priority transactions [OCS07; SMvP08].

Our comprehensive contention management strategy for STM has three main components: (1) lazy (commit-time) acquisition of written locations; (2) extendable timestamp-based conflict detection, in the style of Riegel et al. [RFF07]; and (3) an efficient and accurate means of capturing both priority and conflicts. We introduce two mechanisms—one using Bloom filters, the other using visible read bits—that implement component (3). These mechanisms unify the notions of conflict resolution, inevitability, and transaction retry. They force lower-priority transactions to defer to higher priority transactions only in the presence of actual conflicts. They are orthogonal to the rest of the contention management strategy, and could be used in a wide variety of hardware and software TM systems.

Experimental evaluation demonstrates that (a) the use of a carefully designed lazy STM does not sacrifice performance relative to an eager design; (b) choosing lazy STM is itself an effective contention management strategy, eliminating livelock in practice; and (c) the mechanisms we propose to enable priority scheduling effectively eliminate starvation, at reasonable cost, even for challenging workloads.

We describe our baseline STM system in Section 5.2, focusing in particular on the advantages of lazy acquire. We note that careful data structure design can minimize the cost of the instrumentation necessary for a transaction to read its own writes. As in TL2 [DSS06] and tinySTM [FFR08], we perform fast validation of read set consistency using per-transaction and per-object timestamps. When fast validation fails, we apply tinySTM’s timestamp extension mechanism to avoid aborts whenever possible. We suggest that the poor performance of lazy acquire reported in previous papers may be
due primarily to the use of an STM system without timestamp extension. We describe our use of priority in Section 5.3, and two candidate implementations in Section 5.4. Performance results appear in Section 5.5.

5.2 Baseline Lazy System

High-performance STM implementations must provide both low single-thread latency and good scalability. The first of these goals tends to require low constant overhead on each read or write, while the second focuses on avoiding unnecessary aborts and blocking/waiting. The principal mechanism upon which debate focuses is the manner in which writes are performed. Throughout this section, we refer to those mechanisms that acquire exclusive ownership of to-be-written locations upon first access as “eager”, and those that wait until commit time to acquire ownership as “lazy”. We refer to the corresponding decision to perform writes in-place and use undo logs as “undo”, with “redo” describing systems that buffer speculative writes until commit time.

Our base STM uses lazy acquire, a table of ownership records (1M in our current implementation), extendable timestamps [RFF07], and a hashtable-indexed write set. Prior work has argued that lazy systems are fundamentally slower than eager systems. In this section we demonstrate that two main performance arguments against laziness can be mitigated by careful STM design, and argue that lazy STM has a natural tendency to avoid pathologies that degrade throughput.

Our experiments consider eager acquire with undo logs only; results should extend naturally to eager systems with redo. Additionally, we do not consider mixed invalidation: for workloads with write-write conflicts, mixed invalidation behaves similarly to eager systems with redo. For workloads with read-write conflicts, mixed invalidation...
ation behaves similarly to lazy systems. We also ignore differences in transactional semantics. In particular, weakly atomic eager/undo systems appear to be incompatible with the Java Memory Model prohibition on out of thin air reads [MBS+08a], but are considered here due to their performance. Additionally, while our baseline system is compatible with Moore and Grossman’s type-based semantics [MG08], the separation-based semantics of Abadi et al. [ABHI08], and our own selective strict serializability [SDMS08] (Chapter 7), it does not use TL2-style timestamps, and thus does not implicitly support racy publication [MBS+08a] or programs that exhibit only violation freedom [ABHI08].

All experiments in this chapter were conducted on an 8-core (32-thread), 1.0 GHz Sun T1000 (Niagara) chip multiprocessor running Solaris 10. All benchmarks and STM runtime libraries were written in C++ and compiled with g++ version 4.1.1 using –O3 optimizations. Data points are the average of five 5-second trials, unless we explicitly state otherwise.

5.2.1 Write Set Lookup Latency

In previous lazy STM implementations, the cost of write-set lookups has been high in large transactions. Felber et al. identify lookups as a significant source of latency [FFR08], and show that even in TL2, where a small Bloom filter is used to avoid searching the write set on every read, large write sets quickly saturate the filter and introduce noticeable overhead. We argue that this overhead is not fundamental to lazy STM. Most lazy STM implementations use a linear buffer (a vector) to represent the write set, so after $W$ writes by a transaction, each of $R$ subsequent reads must perform an $O(W)$ lookup in the set, resulting in $O(RW)$ aggregate overhead. We address this overhead by using a hash table to map addresses to indexes in the linear write log, as in
Figure 5.1: Workload with large write sets, modeled as a linked list with 8-bit keys, 95% lookups, and 5% overwrites. The list is pre-populated to store all elements in the range 0–255. Overwrite transactions modify every node in a randomly selected list prefix. Adding a hash table for fast write set lookups eliminates the performance gap between eager and lazy acquisition.

JudoSTM [OCS07]. The hash table keeps versioned buckets which enable $O(1)$ reset, and resolves collisions with linear probing. We rehash whenever the table reaches a 33% load, but do not shrink the table on reset (transaction commit or abort).

Figure 5.1 contrasts this write set with an implementation using a linear buffer (vector) and 32-bit filter. In the benchmark, which is designed to be a torture-test for STM implementations with write set lookup overhead [FFR08], transactions access a sorted linked list holding 256 elements. 95% of transactions are read-only. The remainder are “overwrite” transactions, which write to every node in a randomly selected list prefix. Simply replacing the vector with a hash table raises performance of the lazy STM to the same level as an eager system with undo logs.
In traditional microbenchmarks with small transactions (red-black trees, lists, hash tables), we observed less than 2% overhead when using a hash table instead of a vector to represent the write set. The break even point is dependent on the test platform, the total number of writes, and the distribution of reads and writes. In a RandomGraph benchmark [MSH+06a], where each batch of 2 writes follows a batch of 256 reads on average, the vector can be up to 12% faster on our single-issue Niagara box, but up to 5% slower on a 4-way Intel Core 2. All lazy STM implementations described in the remainder of this chapter use a hash-based write set.

5.2.2 Timestamps and Wasted Work

It has also been argued that lazy STM can scale worse than eager STM due to wasted work [DGK08; FFR08]. Suppose that transaction A writes location L and then concurrent transaction B reads L before A reaches its commit point. If A commits, all instructions issued by B after its access to L constitute wasted work. (Of course, if B reads L before A’s speculative write, and A commits first, all of A’s work will be wasted in either eager or lazy STM.)

What is not as often recognized is that in a system with any significant number of conflicts, there is a good chance that if B aborts eagerly when it discovers its conflict with A, all the work it has done so far will have been wasted if A subsequently fails to commit. Conversely, if B forces A to abort, then all of A’s work will have been wasted if B ultimately aborts. Eager STM typically does not permit the possibility that B could commit before A in this situation, but may allow B to spin in the hopes that A will commit first. In the general case, deadlock avoidance requires that eager STM must abort either A or B when a conflict is detected, even though neither A nor B is guaranteed to succeed.
Evaluation of this fundamental tradeoff (past versus future wasted work) appears to have been clouded by the fact that many comparisons between eager and lazy STM have used TL2 [DSS06] as the lazy representative. While TL2-style timestamps result in low latency, they are inherently pessimistic and reduce scalability by aborting on some easily resolvable conflicts.

With TL2-style timestamps, a transaction $T$ samples the current time $C$ from a global clock at begin time. If it ever encounters a location whose most recent update occurred after $C$, $T$ aborts and restarts. In contrast, with the extendable timestamps of Riegel et al. [RFF07], $T$ re-samples the global clock as $C'$, and then checks that all of its past reads remain valid. If any check fails, the transaction aborts. Otherwise the transaction’s execution is equivalent to one in which all work so far was performed at time $C'$, so the transaction sets its start time to $C'$ and continues. Extendable timestamp implementations may require a handful of additional instructions on every read, resulting in a slightly lower single-thread performance than TL2. In a multithreaded execution, a transaction with extendable timestamps may, in a pathological schedule, be forced to validate on each of $r$ reads. In TL2, the same transaction would abort and restart $r$ times.

Figure 5.2 compares a TL2-like algorithm to eager and lazy alternatives that use extendable timestamps. All threads repeatedly attempt to transactionally insert or remove 8-bit keys in a pre-populated, sorted linked list. All runtimes are implemented in a single C++ framework that isolates the differences in locking strategy and timestamp management. In recent work by the tinySTM group, Felber et al. [FFR08] report similar experiments without the lazy-extendable curve. They suggest that laziness alone (or more specifically, failure to detect conflicts early) is hindering scalability. However,
when we combine lazy acquire with extendable timestamps, scalability is on par with eager acquire.¹

5.2.3 Preventing Aborts

Both TL2 and tinySTM advocate a “Passive” contention management strategy, in which transactions abort when they encounter a lock. The combination of extendable timestamps and lazy acquire, however, suggests a refined strategy (herein called “Patient”, as in “willing to wait its turn”) that reduces the incidence of self-abort.

¹Experiments using the recently-released “commit-time locking” variant of tinySTM, which also uses extendable timestamps, show performance on par with the eager locking variants of tinySTM.
Assuming a lazy STM, a Patient transaction that encounters a lock simply waits (yielding the CPU if there are more active threads than cores). If this lock is the first instance of conflict between the two transactions, then, assuming extendable timestamps, the waiting transaction can simply revalidate and continue once the lock holder commits. Since the waiting transaction is lazy, it does not hold locks and thus its wait cannot block any other concurrent transactions. Moreover aborting the lock holder would not make much sense either: the lock holder must release the lock before the waiting transaction can proceed, and since the lock holder is already in its commit protocol, and about to finish up, aborting it is not likely to make it release the lock much sooner. In fact, it is likely that the lock holder has already committed, and is performing its write-back.

When a transaction reaches its commit point, it attempts to acquire all locks, and aborts if any of them cannot be acquired. This strategy minimizes the window in which a transaction’s completion can cause blocking in concurrent active transactions. It is also most compatible with the use of operating system scheduler hooks to prevent pre-emption when locks are held: there is a good chance a transaction will be able to complete its commit protocol during the scheduler’s “grace period” [KWS97]; executing a sizable chunk of the transaction during that window is much less likely.

In the related context of hardware TM, Bobba [BMV+07] describes three main “performance pathologies” for lazy systems: A long-running transaction may starve in the face of many small writers that commit (“StarvingElder”); the commit protocol may serialize transaction completion (“SerialCommit”); or, when conflicts are high, transactions may convoy at their restart point (“RestartConvoy”) [BMV+07]. Of these three problems, SerialCommit does not apply to our STM, since it does not use a single commit token (note, however, that some variants of RingSTM [SMvP08] and Ju-
doSTM [OCS07] do suffer from this problem). Bobba suggests that a small amount of 
backoff on abort appears sufficient to resolve RestartConvoy. Our priority mechanism 
(Section 5.3) will resolve StarvingElder, and all starvation.

5.2.4 Approaching Livelock Freedom

Unlike the hardware TM in Bobba’s study, our design uses invisible reads and can 
admit livelock. While our mechanisms for preventing starvation will naturally prevent 
livelock (by ensuring that someone commits), we observe that, in practice, lazy STM 
avoids livelock as well. The intuition is that, by design, a transaction holds locks only 
for a brief period of time. Livelock occurs when lock-holding transactions cause each 
other to abort. Most aborts occur when a still-active transaction $A$ discovers that some 
other transaction $B$ has modified a location that $A$ has already read or written. Gener-
ally this means that $B$ has made progress. There are three potential exception cases.

The first case arises when $B$ is part-way through its commit protocol, but has not 
yet committed. $A$’s abort will prove unnecessary if $B$ subsequently fails to commit. But since $B$ is already in its commit protocol, it must abort for a different reason than $A$. So this scenario alone cannot lead to livelock.

The second case arises when transactions have multiple write-write conflicts, and 
are all in the lock-acquisition phase of the commit protocol. If locks are acquired in 
sorted order, one of the transactions is guaranteed to acquire all of its locks and progress 
to validation, while the others will abort. Sorting takes time, however, and most im-
plementations do not bother. As described previously, our STM releases all locks and 
aborts when a held lock is encountered during acquisition. This admits the possibil-
ity that two committing transactions will abort because of each other. If this repeats, 
livelock is possible. Given the narrow width of the commit protocol, however, and
natural fluctuations in system timing, repeating mutual aborts would seem exceedingly unlikely. Should they become an issue, they could easily be addressed (as by Bobba et al.) with randomized abort-time back-off.

The third case arises when transactions have symmetric read-write conflicts, and are in the validation phase of the commit protocol (after acquiring locks). Transaction $A$, for example, may have read location $L$ and written location $M$, while transaction $B$ has written $L$ and read $M$. During validation, $A$ detects its read-write conflict first, but before it can unlock its locations, $B$ detects its read-write conflict and also aborts. This multiple-abort scenario can also occur when one of the read-write conflicts is promoted to a write-write conflict, as when $B$ writes both $L$ and $M$.

The odds of this third case are also very low. Our system uses a global commit counter [SMSS06], allowing a transaction to notice when no one has committed during its execution, and to commit itself without performing validation, so this livelock condition will never be seen in two-transaction situations because only one transaction will be validating at any point in time (triggered by the other’s optimized commit). With more than two transactions, the likelihood that multiple transactions (1) have symmetric read-write conflicts, (2) repeatedly validate at exactly the same time, and (3) fail to release locks and abort quickly enough to avoid mutual aborts, seems to be vanishingly small, and can be further avoided with randomized abort-time back-off. In repeated experiments, we have been unable to devise any workload, no matter how contrived, that will livelock a lazy STM with extendable timestamps even without the backoff.

For eager STM, we have found that the Passive contention management policy (self-abort on conflict) is adequate when the memory access pattern is fairly regular. In Figure 5.3, for example, all transactions start at the head of a list and progress forward. Unfortunately, once the access pattern becomes irregular (for example, when transac-
Figure 5.3: RandomGraph benchmark. The blocking “Passive” strategy avoids livelock. Polite and other nonblocking strategies fail to prevent livelock for eager STM with invisible reads.

transactions can start at either end of a doubly linked list), eager STM is prone to livelock, and even though throughput remains good for lazy STM, starvation is likely (Figure 5.4).

A potentially more troubling example appears in Figure 5.5. Here each transaction performs 8 tree operations in a forest of 5 trees. Tree operations can occur in any order, and different trees have different key ranges and lookup ratios. Measurements of a single-tree workload show that eager STM with Passive contention management performs well. Eager also performs well for multiple trees under low thread counts, but suffers dramatically with larger counts. In effect, for this benchmark, eager STM fails to provide the “performance composability” that one would hope to achieve in any STM system. Instead of scaling, the eager runtime experiences a dramatic increase in aborts, as shown in Figure 5.5(b).
(a) Throughput. Even though there are no write-write conflicts, eager acquire livelocks.

(b) Commit rate (16-threads). Eager threads receive relatively fair slices of an almost nonexistent pie.

Figure 5.4: Pathological microbenchmark. Transactions traverse a doubly-linked list in different directions, reading all entries and modifying 8 entries.
Figure 5.5: Forest microbenchmark with one low-contention tree (80% lookups, 20-bit keys), three medium-contention trees (50/50 insert/delete ratio, 8-bit keys), and one high-contention tree (50/50 insert/delete ratio, 4-bit keys).
5.3 Defining Priority

The STM of Section 5.2 provides good throughput and generally avoids livelock. However, it can experience starvation, as shown in Figure 5.4(b). The experiment runs for a fixed time interval, and all transactions conflict. While the lazy runtimes maintain good throughput, they do so without any concern for fairness. Instead of providing an even 6.25% of the total commits, some threads perform much more than their fair share (up to 25%) while other threads perform as little as 1%. The apparent fairness of the eager runtime is an implementation artifact: the entire benchmark is livelocked, each thread is guaranteed to commit once, and there are only 20 total commits for the 5-second, 16-thread execution.

In this section we describe the requirements for a user-level priority mechanism, and then show how that mechanism can be augmented to transparently prevent starvation and provide fairness, while also unifying priority with retry-based condition synchronization and inevitability.

5.3.1 Fairness

There is no agreed-upon definition of fairness for concurrent workloads. Certainly it is not desirable for transactions to starve; however, only the programmer can determine which transactions are relatively more important than others. Thus we use programmer-specified priority as the principal mechanism for selecting a fair schedule for transactions. All transactions have zero priority unless explicitly marked otherwise. If the programmer assigns transaction \( H \) higher priority than transaction \( L \), then the runtime should ensure that \( L \)’s writes do not prevent \( H \) from committing, most likely by preventing \( L \) from committing. However, we agree with Gottschlich and Connors [GC08]...
that if \( L \) and \( H \) do not conflict, then \( L \) should not be prevented from committing when \( H \) is active. The requirement that \( L \) can commit so long as it does not conflict with \( H \) is a departure from most previous work on starvation avoidance. Previous proposals either promoted \( H \) to an inevitable state \([OCS07; SSD^{+}08; WSAT08]\), in which it cannot abort, or else serialized transaction commit \([SMvP08]\). Serialization can avoid starvation entirely, but it does so at the expense of any write concurrency in systems with invisible readers: once a starving transaction is identified and given priority, other transactions, even if they are nonconflicting, cannot commit unless they are read-only; allowing writers to commit could otherwise introduce conflicts with the priority transaction’s reads.

If \( L \) and \( H \) have the same priority, we do not require the runtime to provide any guarantees about the order in which the transactions commit. More importantly, if a workload admits an infinite set of transactions \( \mathcal{K} \) of equal priority, our default model of priority does not guarantee that every \( K \in \mathcal{K} \) will eventually commit; some may starve if there are continuous conflicts with concurrent, same-priority transactions that succeed. In other words, we do not guarantee fairness among equal priority transactions. This choice is made for the sake of practicality; we do not contend that it provides ideal semantics. To avoid starvation, we will actively manipulate priorities.

The main challenge to priority-based conflict resolution is the lack of read visibility \([GC08; GHP05b]\). In order to prevent a low priority write from interfering with a high priority read, the low priority transaction must somehow be aware of the read. This property is not typically provided in STM, due to the overhead (cache ping-ponging in particular) of visible reader lists \([SATH^{+}06; MSH^{+}06a]\).\(^2\) However, since our baseline STM provides good throughput without user-defined priority, we need only provide

\(^2\)We observed this overhead in the Vis/Eager curves of Figures 2.4–2.5, pages 54–55.
read visibility for transactions with nonzero priority. When no such transactions exist, the runtime should incur only a small constant overhead at commit time. Only when transactions with nonzero priority are in-flight should additional overheads apply.

In a lazy STM, priority-related overhead occurs at only three points. First, a priority transaction must make its reads visible in some fashion, incurring constant overhead on each of \( R \) reads. Second, when there exist any priority transactions, committing writers must ensure that their commit would not invalidate concurrent higher-priority reads. In a lazy STM, this test is performed after acquiring all locks, and for a transaction with \( W \) writes, should introduce no more than \( O(WP) \) overhead, where \( P \) is the number of higher-priority transactions. If priority conflicts are detected, the committer can either release all locks and wait, or else abort. The latter option is simpler, and aborting in this situation meshes well with a Karma-like mechanism to avoid starvation by automatically elevating priority after consecutive aborts. Third, when an in-flight priority transaction \( H \) detects a conflict with a lower-priority lock holder \( L \), it must wait for the lock holder to release the lock. If \( L \) acquired all locks before \( H \)’s first access to the location, \( L \) can commit, at which point \( H \) will reload the lock covering the conflicting location, and extend its timestamp. If \( L \) acquired all locks after \( H \)’s first access to the location, then \( L \) will detect \( H \)’s access and abort. In either case, the delay for \( H \) should be minimal.

### 5.3.2 Supporting Inevitability

A program might require transactions to perform I/O or other irreversible operations. The underlying mechanism for these must-be-inevitable transactions requires that (1) at most one transaction is inevitable at any time, and (2) no concurrent transaction may cause an inevitable transaction to abort [SMS08]. When inevitability is used
to call un-instrumented, precompiled binaries, or to make system calls, it may also re-
quire that (3) all writes are performed in-place [SMS08; SSD+08; WSAT08], which
usually also requires encounter-time (eager) locking for the inevitable transaction only.
The first two of these conditions are easily expressed in our priority framework, and
suffice for systems without precompiled libraries: we need only require that the high-
est priority level can be assigned to no more than one active transaction. Support for
encounter-time locking and in-place update by the inevitable transaction, if needed,
must be provided by the STM runtime. As long as the runtime and priority manager
agree that a transaction is inevitable, this requirement is straightforward to implement
using existing techniques.

5.3.3 Integrating Retry

Typically, programmers perform transaction condition synchronization via a retry
mechanism. With retry, an in-flight transaction determines, through reads of shared
memory, that a precondition for its execution does not hold. At this point, the transac-
tion requests that the runtime abort it, and not reschedule it until some location in its
read set is modified by another transaction [HMPH05; SSS08]. In our priority frame-
work, these transactions are assigned negative priority once they call retry. Once neg-
ative priority is assigned, a transaction makes all of its reads visible, double-checks its
read set validity, and then waits for notification that one of its reads has been invali-
dated. Upon wakeup, negative priority is discarded. Since we use lazy acquire, when
these transactions are restarted they will not hold locks before reaching their commit
point, which prevents retryers from blocking concurrent transactions that might actu-
ally satisfy the condition on which transactions are waiting (a useful property since
retry admits spurious wakeups). Furthermore, since retryers have negative priority, no concurrent transaction is required to negotiate when writing to locations they have read.

5.3.4 Automatic Priority Elevation

When there are no user-specified priorities, transactions default to a mode in which no priority overhead is incurred. However, as noted in Section 5.2.4, this mode admits the possibility of livelock (in theory) or starvation (in practice). Likewise, at any priority level, some transaction may starve while others commit, as discussed in Section 5.3.1. Given an implementation of priority, however, a simple Karma-like mechanism suffices to break starvation. Inasmuch as livelock is the condition of all transactions starving, our Karma mechanism also addresses livelock.

Starvation detection is quite simple: a transaction need only count its consecutive aborts, and use that value as its Karma. We propose two possible mappings of Karma to priority. With a static mapping, for some user-specified constant \( X \) we define priority as \( \lfloor \frac{\text{Karma}}{X} \rfloor \), that is, for every \( X \) consecutive aborts, a transaction’s priority increases by 1. With load-sensitive mapping, for some constant \( X \) and transaction count \( T \), a transaction’s priority is \( \lfloor \frac{\text{Karma}}{XT} \rfloor \). This mapping captures the intuition that a transaction that takes \( X \) times as long as the next longest-running transaction could expect in the worst case to observe aborts proportional to both the amount of concurrency and the difference in transaction length. An abort rate above this level can then be considered unfair (or possibly starvation). To compose Karma-based priority elevation with user-requested priority, we simply raise a transaction to the sum of its Karma priority and its requested priority. Once a transaction succeeds, its Karma is discarded.

While even the composition of Karma, priority, and randomized exponential backoff on abort do not lead to a provably livelock-free or provably starvation-free system,
we expect in practice to avoid both pathologies. Furthermore, combining exponential
backoff with priority-based starvation avoidance should resolve all of the “pathologies”
identified for lazy hardware TM by Bobba et al. [BMV+07].

5.4 Implementation

In this section we present two independent mechanisms to implement priority. Both
mechanisms use the same interface, presented in Listing 5.1, to interact with the STM
runtime library. Below we briefly discuss how priority is supported via this interface.
For simplicity, we consider only those situations where inevitability or priority is as-
signed before any transactional work is performed. Extending this interface to support
dynamic changes to priority (to include inevitability) is straightforward [SSD+08].

Our mechanisms require that transactions with nonzero priority make their reads
visible to concurrent transactions. We use Bloom filters and RSTM-style visible reader
bits [MSH+06a], respectively, to provide this visibility. In both cases, the mecha-
nism for read visibility is implemented within the priority manager, and completely
transparent to the underlying STM. In particular, in contrast to RSTM, the visible
reader bits are not part of the usual per-location metadata, and need not even be al-
located at the same granularity. We leave as future work the various ways in which
the priority manager might be specialized to an STM implementation, such as by pass-
ing RingSTM’s Bloom filters to the preCommit, postCommit, and preRetry
methods [SMvP08], or by using tinySTM ownership record addresses as parameters to
preOpen and preRetry.

Prioritizing Reads The requestPrio() method attempts to raise a transaction’s
priority by first reserving a visible read handle, and then associating that handle with
the requested priority. Every subsequent read must then be preceded by a `preOpen()` call. When a transaction with nonzero priority calls `preOpen()`, the priority manager makes the caller a visible reader of the location. The STM is responsible for blocking after this call if the location is locked, and extending timestamps accordingly once the location is released. When write-read ordering is required by the processor memory model, it is provided within the call. Unlike previous work, this approach incurs no overhead in the common case. In contrast, Scherer and Scott’s policies [SS05a] require statistics gathering on every access. The Greedy policy [GHP05b] requires every read to be visible, even for transactions with no priority, in workloads that do not exhibit fairness pathologies.

**Prioritizing Writes** Since most writes follow reads to the same address, the STM can call `preOpen` on every write, as well as every read. While not strictly necessary, this decision simplifies many code paths in lazy STM implementations that use extendable timestamps.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void <code>onBegin()</code></td>
<td>sets priority based on Karma</td>
</tr>
<tr>
<td>bool <code>tryInevitable()</code></td>
<td>request MAX_PRIO</td>
</tr>
<tr>
<td>void <code>requestPrio(level)</code></td>
<td>attempt to raise priority</td>
</tr>
<tr>
<td>void <code>preOpen(addr)</code></td>
<td>mark priority on address</td>
</tr>
<tr>
<td>void <code>onAbort()</code></td>
<td>backoff or yield, increase Karma, and then unmark priority</td>
</tr>
<tr>
<td>bool <code>preCommit(wset)</code></td>
<td>return false if there exists a higher priority transaction with a conflicting read</td>
</tr>
<tr>
<td>void <code>onCommit()</code></td>
<td>clear Karma, unmark priority</td>
</tr>
<tr>
<td>void <code>preRetry(rset)</code></td>
<td>mark priority on read set</td>
</tr>
<tr>
<td>void <code>postRetry()</code></td>
<td>wait for notification, then unmark priority</td>
</tr>
<tr>
<td>void <code>postCommit(wset)</code></td>
<td>wake any retryer whose rset overlaps wset</td>
</tr>
</tbody>
</table>

Table 5.1: Interface between priority manager and STM library
Detecting Priority Conflicts  A committing transaction could inspect every location after acquiring its lock, to identify all concurrent prioritized readers. We prefer, however, to acquire all locks and then perform a single call to identify conflicts between the committing writer and concurrent, higher-priority readers. In the common case, this call finds that there are no priority transactions and does no further work. When there are priority transactions, deferring detection of priority transactions until after all locks are held either enables us to work with a single summary of the write set (in the Bloom filter mechanism) or filter multiple potential conflicts (in the visible reader bit mechanism), saving substantial work.

Inevitability  Inevitability support is provided through a custom implementation of the requestPrio() method, which ensures at most one transaction holds maximum priority at any time.

Retry  To retry, the STM first calls preRetry() to make all reads visible, then validates (to avoid a window in which the only transaction that could wake the retryer commits and invalidates the retryer’s read set [SSS08]). If the validation succeeds, the runtime then calls postRetry() to force the thread to wait on a semaphore. As with priority, we maintain a count of the number of retrying transactions. Once a writer commits and releases its locks, it calls postCommit() to wake any retryers whose reads it invalidated. When there are no retryers, this call does no further work. When retryers exist, the implementation of postCommit closely resembles preCommit, except that it searches for conflicting retryers rather than conflicting higher-priority transactions.
Karma  The onBegin, onAbort, onCommit, and preRetry calls manipulate a consecutive abort counter, which is used to compute Karma. When Karma is used for priority elevation, the requestPrio method adds Karma-based priority to the request, and uses the sum as the new transaction priority. Once the transaction completes, the abort counter is reset, which discards any accumulated Karma.

### 5.4.1 Priority Read Bits

Our first mechanism for priority adapts the visible reader bits employed by the RSTM algorithm [MSH+06a]. The mechanism maintains an active transaction table (ATT) with $T$ entries, each corresponding to an active transaction with nonzero priority. It also maintains $L$ reader records (rrecs), and uses a hash function to map addresses to rrecs, much like the mechanism used to map locations to ownership records in STM. Each rrec is a $T$-bit array, where each bit corresponds to an entry in the table of active transactions.

The requestPrio() method succeeds only if the transaction (call it $R$) is able to identify an unreserved element $I$ in the ATT and atomically point that element to itself. If the call succeeds, a subsequent call to preOpen() first locates the rrec corresponding to the input address, and then atomically sets the $I$th bit of that rrec. $R$ also adds the rrec to a transaction-local log, so that it can clear the $I$th bit when it ultimately commits or aborts. If $R$ is unable to reserve an entry in the ATT, it still records its priority, in the hope that it can run to execution without protecting its reads with rrec updates. If it cannot, it will attempt to reserve an ATT entry each time it restarts, until it either reserves an entry or commits. As we shall see below, by incrementing its priority, the caller enables itself to avoid incorrect aborts due to conflicts with same- or lower-priority transactions that successfully reserved ATT entries.
The `preCommit()` call takes as a parameter the write set whose corresponding locks were just acquired by the calling transaction. For each address written, the corresponding rrecs are unioned into a single summary rrec. For \( W \) writes, this introduces \( O(WT) \) overhead. After the summary rrec is built, each bit is tested; if any bit \( i \) is nonzero, and the \( i \)th element of the ATT represents a transaction with higher priority than the caller, then `preCommit()` returns false and the caller aborts. This step takes at most \( O(T) \) time. Aborting lets the caller defer to the higher priority transaction while simultaneously incrementing Karma, to avoid starvation.

The wakeup phase of our retry mechanism works almost identically, though conflict tests are performed only after the writing transaction completes. If both priority and retry transactions exist, the summary rrec created by `preCommit()` can be recycled; otherwise, a new summary must be created. Then, each bit is tested, and any bit corresponding to an ATT entry with \(-1\) priority is awoken. Retryers, for their part, simply mark all rrecs corresponding to locations they have read in `preRetry()`. As with the `requestPrio()` call, if a retryer is unable to reserve an ATT entry, it does not mark its reads. A subsequent call to `postCommit()` will not wait on a semaphore, but rather call `usleep()` to yield briefly and then restart.

Our rrecs offer three significant advantages over traditional list-based visible readers. First, the use of a global table of rrecs avoids any need for dynamic memory allocation. Second, the rrecs covering multiple locations can simply be unioned to create a complete list of potentially conflicting transactions, with no duplicate entries. List-based implementations typically lack such low-overhead filtering. Third, for large values of \( T \), SIMD instructions can be used on many architectures to accelerate the creation of the summary rrec. Since locations map to rrecs by a hash function, false conflicts are possible; however, these false conflicts will never impede progress for
high-priority transactions. Instead, false conflicts may cause increased aborts in low-
priority writers and increased wakeups of retrying transactions.

While we prefer the use of a single ATT with wide (128 bit or more) rrecs, it is
possible to support multiple ATTs, corresponding to different ranges of priority. ATTs
could then be configured with different rrec counts $L$ and different rrec widths $T$. A
particularly appealing option is to maintain two ATTs – one for the inevitable transac-
tion and one all other priority transactions. This design point eliminates the need for
atomic updates to the inevitability rrecs: since only one transaction can be inevitable,
only one transaction can modify the set of “inevitability” rrecs, and it can do so with
a normal store followed by a write-before-read memory fence. On processors whose
memory fences are cheaper than read-modify-write operations, this option is particu-
larly appealing.

### 5.4.2 Priority Read Filters

Our second mechanism for priority uses Bloom filters [Blo70] to make reads visi-
ble. Unlike priority read bits, this mechanism has no fixed memory footprint. Instead,
it maintains a list of transactions with nonzero priority. Each list entry consists of a ref-
ference to the corresponding transaction, and a Bloom filter representing the addresses
passed by that transaction to $\text{preOpen}()$.

The $\text{requestPrio}$ method is far simpler with this implementation: it simply
appends an entry to the list (or, if list entries are never removed, activates the disabled
list entry corresponding to the calling transaction). Calls to $\text{preOpen}()$ mark the bits
in the filter corresponding to the address being read by the priority transaction, and then
perform a write-before-read memory fence.
At commit time, transactions scan the global list for entries corresponding to higher-priority transactions. For each entry, the transaction must check if any address in its write set is stored in the filter. When filters are configured with a single hash function, this operation can be heavily optimized. The `preCommit()` function first constructs a filter representing the caller’s entire write set. It then intersects this filter with the filter corresponding to any higher-priority active transaction’s reads. If an intersection is nonzero, the calling transaction aborts. This optimization results in $O(W)$ overhead to construct the committer filter, and then with $N$ concurrent transactions, at most $O(N)$ overhead to perform the intersections (assuming the size of the filter to be constant). With multiple hash functions, there is $O(WN)$ overhead to detect conflicts with concurrent priority transactions. Throughout our analysis, we treat the cost of filter intersection and clearing as constants; with SIMD instructions these constants can be kept relatively low, allowing larger filters which decrease the false conflict rate and lower susceptibility to the “birthday paradox” [ZR07].

Again, retry wakeup works in the same manner as `preCommit()`, though conflict tests are performed after the writer commits. The write-set filter constructed to detect conflicts with priority readers can be reused to identify which retryers to wake. Furthermore, `preRetry()` does not require write-before-read ordering after every filter update, only after the last update.

It is possible to vary the filter size and set of hash functions for different priority levels, and to maintain multiple lists storing filters of different priority ranges. However, these refinements risk introducing more overhead to maintain the lists of transactions with nonzero priority. As presented above, our mechanism is simple, amenable to SIMD optimization (both for clearing and for intersection), and requires only one pass through a transaction’s write set for all retry and priority conflict detection.
Relaxing Memory Fence Requirements  For all priority levels below the maximum (inevitable) level, we assume it is acceptable for conflicts to occasionally be won by lower priority writers. Given this relaxation, we can eliminate the write-before-read memory ordering for insertions into the filter for priority (but noninevitable) transactions. With this optimization, the read filter updates may fail to propagate to a concurrent, lower-priority writer, in which case the higher priority transaction must abort and restart. In practice, we expect such aborts to be rare. In the subsequent evaluation, we refer to results using this optimization as “no WBR”.

5.5 Evaluating Priority Mechanisms

In previous sections, we demonstrated that lazy STM can avoid aborts, avoid livelock, and provide good throughput, especially in situations where eager STM experiences pathological behavior. These results did not, however, require our priority mechanism; they only required a carefully engineered lazy STM. We now turn our attention to the effectiveness of our policies in avoiding starvation. All results were collected on the Niagara platform described in Section 5.2.

5.5.1 Breaking Write-Write Starvation

Figure 5.6 presents another pathological microbenchmark using a doubly-linked list. In this workload, half of the threads use transactions to forward-traverse the list; the remainder perform reverse traversal. Every transaction attempts to increment a counter in each node of the list.

Naturally, the benchmark does not scale, and as threads are added, we expect contention for global variables to cause some degradation. However, in the “Base” bars of
Figure 5.6: The cost of priority-based starvation freedom in a pathological microbenchmark. Transactions write every entry of a 256-element doubly-linked list, traversing in forward or reverse order. Transactions increment priority on 16 consecutive aborts.
Figure 5.6, we see that threads commit in highly unequal proportion. In some runs, a single thread may perform 30% of total commits, while other threads do not commit even one transaction during the 5-second experiment.

In the Bloom, Bloom/no WBR, and VisRead curves, transactions use a priority-based contention manager with Karma to break starvation. For every 16 consecutive aborts, a transaction increments its priority. As these mechanisms introduce additional latency (particularly on the single-issue cores of our test platform), resorting to priority degrades throughput up to 50%. However, Karma also increases fairness. While individual threads may commit up to twice as often as each other, no thread has less than 4% of the total commits (the expected average is 6.25%). In contrast, without priority some threads do not commit at all. Visible reads appear particularly fair, though this observation may be tainted by their lower throughput and the fact that commit rates are a snapshot for a single 16-thread execution. Some of this lost throughput can be restored by changing the Karma parameters, using a load-sensitive mapping of aborts to Karma, or adding randomized exponential backoff on abort.

We also note that avoiding memory fences accelerates priority transactions, thereby raising throughput, without decreasing fairness. Writes propagate quickly enough that the relaxed ordering is unlikely to result in false conflicts. At the same time, fences are relatively inexpensive on the Niagara processor, and the single-issue pipeline means that the computational part of the Bloom filter update is on the critical path of every read. As a result, the latency reduction of elided fences is minimal. On a SunFire 880, where fences are still fairly expensive, excess issue width allows us to soak up much of the rest of the update cost, and eliding the fence reduces the cost of instrumentation by 15–25%.
5.5.2 Read-Write Starvation

To evaluate read-write starvation, we consider an extensible hash table. The table is created by a factory, which registers each constructed table in a private list. The factory also runs a management thread, which polls every constructed table, and rehashes whenever a table exceeds some threshold (in our experiments, rehashing occurs when some bucket contains more than 4 elements). On every put operation, transactions estimate the depth of the bucket into which the put is performed. If the depth exceeds some boundary, the put operation also updates a flag indicating that the table requires rehashing.

The management thread checks the flag of every table, in a single transaction. If no table’s flag is set, the thread uses retry to sleep until some flag is set, indicating that some table requires rehashing. Whenever a set flag is encountered, the management thread immediately exits the polling transaction, rehashes the table, resets the flag, and then starts a new polling transaction to test all flags. Using retry in this manner ensures that the rehash thread does not consume CPU resources when it is not actively rehashing. In our experiments, retry is infrequent enough that we cannot measure the difference between visible read bits and Bloom filters.

We consider three approaches to rehashing. First, with inevitability, the rehash thread can avoid the latency of write buffering or undo logging, as it is given infinite priority and guaranteed that it will not abort. Second, with high but not infinite priority, these guarantees are not provided, but the rehash thread still is unlikely to starve. Neither inevitability nor priority forbids concurrent read access. Third, we consider rehashing via privatization [MBS+08a]. This idiom effectively locks the table, reads it nontransactionally during rehashing, and then unlocks it with a “publishing” trans-
action. Concurrent puts that encounter the lock use retry to avoid unnecessary waiting.

In Figure 5.7, we consider a workload where each transaction performs 8 puts into the hash table, and put transactions run with fixed priority 0. The ideal curve uses a table that is initially very large, and which does not require rehashing. All other curves use an initial table with 8 buckets. Each test performs a total of 64K transactions, divided among the active threads. The degradations at 8 and 16 threads reflect the fact that the Niagara processor has 32 thread contexts, but only 8 cores; analogous though less pronounced effects can be seen in Figure 5.4(a) at 8 and 16 threads.

If the rehash thread also runs with priority 0, it starves (not shown), and the benchmark takes more than 30 seconds to complete. Worse yet, the benchmark then fails.
to scale at all, as most transactions conflict on at least one bucket. With any rehash mechanism that does not starve, we see strong scaling.

From a semantic perspective, priority is clearly the preferred approach to extensible hashing. Privatization (locking) decreases concurrency by forbidding all access to the table during rehashing. Depending on the mechanism used [SMS08], inevitability may jeopardize concurrency in other ways. Suppose the benchmark also included a thread that used transactions to perform I/O. Since inevitability is restricted to one transaction at a time, the arbitrary decision to use it for rehashing (as a performance optimization) would prevent a concurrent transaction from using it where semantically required. Additionally, if a high-priority request to write to the table arises during rehashing, we would really like the rehash to be abortable. Furthermore, inevitability may preclude composing the rehashing code within another transaction [SKBY07; WSAT08].

Fortunately, priority also offers competitive performance, matching the throughput of inevitability and performing better than two of the three privatization options. Visible reads appear to perform slightly worse than Bloom filter-based priority. This may, however, be an artifact of filter and read bit configuration. We maintain an array of 1M 128-bit visible reader bits, but use only 1024-bit filters. Thus if higher-priority execution represented a larger fraction of total execution, our filters would cause more false conflicts than the read bits. Increasing the filter size to improve scalability would also reduce performance, especially without SIMD instructions.

The variation among privatization options relates directly to the implementation of retry. The default “Privatize” curve assumes no retry support, and instructs retryers to call \texttt{usleep(50)} (50\,\mu s is the shortest observable yield interval supported by the OS). The “Privatize+Bloom” and “Privatize+VisRead” curves implement \texttt{retry} as negative priority using Bloom filters and rrecs, respectively. As in Figure 5.6(a), the
lower overhead of the Bloom filter mechanism gives it an advantage relative to the visible reader implementation. The advantage is exaggerated since all retryers wait on the same location, resulting in cache thrashing as they atomically update the same rrec in order to become retryers, and then to release their retry status.

5.6 Summary

The primary motivation for TM is to provide a simple way for programmers to write correct, scalable parallel programs. For a TM system (implemented in hardware or software) to satisfy this goal, it must provide some guarantees about progress. However, many runtimes allow pathological slowdown relating both to performance composability and conflicts induced by the workload. In the worst case, these conflicts can result in livelock and starvation.

Our comprehensive strategy for contention management in software transactional memory avoids livelock and starvation. It uses (1) lazy acquire, (2) extendable timestamps, and (3) an orthogonal mechanism to capture the read sets of transactions with non-default priority. We presented two different implementations of the priority mechanism, one based on Bloom filters, the other on visible read bits. These implementations allow us to unify user-defined priority, retry-based condition synchronization, and inevitability under a single mechanism orthogonal to the underlying STM.

In this chapter, we did not provide a direct comparison with systems using mixed invalidation (Chapter 2). However, our evaluation provides insight into the benefits and weaknesses of mixed invalidation. Since mixed invalidation defers detection of read-write conflicts until commit time, it avoids pathological behavior due to read-write conflicts. Thus it is likely that mixed invalidation would not degrade in the performance
composition workload (Figure 5.5(a)), and that it would not livelock for the experiment in Figure 5.4(a). However, in Figure 5.6(a), where there are write-write conflicts, mixed invalidation attempts to resolve those conflicts early. Consequently, we expect it to behave similarly to eager acquire—that is, to be prone to livelock and to exhibit extreme sensitivity to contention management policies. Shriraman and Dwarkadas have recently come to the same conclusion for mixed invalidation in HTM [SD09].

Our strategy stands in sharp contrast to early work on contention management policies [GHP05b; SS05a]. That work focused on eager acquire, where livelock is a serious problem. It was also evaluated on systems whose comparatively high total overhead tended to mask the incremental cost of “hooks” for contention management. Our work focuses on lazy acquire in a highly optimized runtime. We argue that extendable timestamps make lazy acquire competitive with eager acquire for “well behaved” applications. Further, for “poorly behaved” applications, the narrow commit window of lazy acquire allows it to dramatically outperform eager acquire, effectively eliminating livelock as a practical concern. In this context, an extremely simple “Patient” policy (wait until the lock holder gets out of the way) minimizes unnecessary aborts. Starvation and priority can then be handled safely with a separate mechanism.
6 Inevitability Mechanisms for Software Transactional Memory

The previous chapters of this dissertation focused primarily on performance, specifically lowering single-thread latency and improving throughput. One of our central claims is that transactions will not be useful to programmers unless they allow most of the operations that are typically performed in critical sections. In particular, since most I/O operations cannot be rolled back, it is tempting to forbid their use within transactions.

In Chapter 5, we began to consider a solution to the problem of I/O in transactions by briefly discussing a technique that unified inevitable transactions with user-specified priority. We now explore this option in greater depth. We first describe the requirements that a runtime must provide in order for a transaction to be guaranteed not to abort. We then propose a variety of mechanisms that permit some transactions to run in parallel with an inevitable transaction. These mechanisms vary in terms of the latency experienced by the inevitable transaction, the types of transactions that can commit during the inevitable transaction’s execution, and the delay that transactions experience at their boundaries (and when becoming inevitable) to support each mechanism. We also
show that inevitability can be used as a simplistic contention management policy, and
to improve the speed of certain common-case transactions.

6.1 Introduction

Even the most scalable STM implementations bear considerable runtime and se-
monic overhead. An STM must instrument program loads and stores, and may require
up to $O(n^2)$ time and $O(n)$ space overhead to ensure the correctness of a transaction,
where $n$ is the number of locations accessed. Furthermore, any TM that permits op-
timistic concurrency must provide a rollback mechanism to resolve conflicts. Since
rollback can happen at any time during the execution of a transaction, the program-
er cannot safely perform operations that have irreversible side effects—e.g., I/O and
system calls—from a transactional context. In a library-based STM, it is also unsafe
for transactional code to call precompiled “black box” binaries that may access shared
locations, unless binary rewriting is available [FFM+07; OCS07; YWWJ06].

A straightforward solution to these problems is to identify certain inevitable transac-
tions, at most one of which can be active at a time, and to ensure that such a transaction
never aborts. Once rollback is prevented, an inevitable transaction may safely perform
I/O, syscalls, and other irreversible operations without concern that they will ever need
to be “undone”. It may also be able to skip certain bookkeeping operations associated
with rollback, allowing it to run faster than an abortable transaction—an observation
that raises the possibility of judiciously employing inevitability in the underlying sys-
tem as a performance optimization for transactions that don’t actually require its se-
ematics, or to avoid starvation in runtimes that do not follow the design we proposed in
Chapter 5.
6.1.1 Inevitability in Existing TM Systems

The original TCC proposal [HWC+04], which serialized concurrent transactions at their commit points, provided a notion of “age” through which inevitability could be provided; in effect, the active transaction that had started earliest in time was guaranteed to commit. TCC’s age-based inevitability served both to prevent starvation and to allow irreversible operations inside transactions: to perform such operations, the transaction waited until it was “oldest” and then could proceed safely. Subsequent discussions of inevitability tend to focus on this design point [BZ07; HPR+07; SKBY07], and assume that an in-flight inevitable transaction must prevent all concurrent transactions from committing.

In software, this design is exemplified by JudoSTM [OCS07], where all concurrent transactions block at their commit point when an inevitable transaction is in-flight. While this permits some concurrency (a transaction can execute from its begin point up to its commit point during the inevitable transaction’s execution), it does not allow any non-inevitable transaction to commit until the inevitable transaction completes. Moreover several features of JudoSTM, including dynamic binary rewriting and value-based conflict detection, limit the applicability of their method to other TM designs.

To the best of our knowledge, there are only two exceptions to what we might call the “commit-token model” of inevitability. The first is the “unrestricted” hardware TM proposed by Blundell et al. [BLM06], which allows non-inevitable transactions to commit alongside an inevitable transaction. This system exposes the inevitable status of a transaction to the hardware, and leverages the fact that the cache coherence protocol can identify conflicts between inevitable reads and non-inevitable writes. Conflicts between an inevitable transaction and concurrent non-inevitable transactions are resolved by favoring the inevitable transaction; non-inevitable transactions are allowed to
commit when no such conflicts are detected. The second exception is the “irrevocable transaction” proposal by Welc et al. [WSAT08], which closely resembles our Inevitable Read Lock mechanism (Section 6.3.4), and was developed at roughly the same time as the mechanisms presented in this chapter.

### 6.1.2 Contributions

This chapter presents five mechanisms that allow non-inevitable read-only transactions to commit while an inevitable transaction is active. Two of these mechanisms also allow non-inevitable writer transactions to commit. These five approaches to inevitability are largely orthogonal to the underlying TM algorithm. Any STM that provides versioning and ownership through metadata should be able to use our mechanisms with little modification, and we believe that hardware and hybrid TMs could support some of these mechanisms with little modification.

Through a series of microbenchmarks, we demonstrate that our mechanisms introduce very little latency and allow high levels of concurrency. We also assess the orthogonal question of whether inevitability is a reasonable mechanism for enhancing performance. We consider a workload characterized by a large pool of tasks, each represented by a transaction, where tasks do not synchronize with each other. In such a setting, we find that inevitability can improve throughput so long as transactions are not trivial, but the effect decreases as parallelism increases.

### 6.2 STM Support for Inevitability

Most STM runtimes control concurrent access using orecs located either in object headers or in a hash table indexed by data address. Intel’s McRT [SATH+06]
and Microsoft’s Bartok [HPST06] use the metadata to lock locations on first access and make modifications in-place. LibLTx [Enn06] also locks locations on first access, but buffers updates and only makes those updates visible upon successful commit. TL2 [DSS06] and some variants of TL [DS07] and RSTM [SSD+07] buffer updates, and only lock locations at commit time. Other STM algorithms, such as DSTM [HLMS03], OSTM [Fra03; FH07], ASTM [MSS05], RSTM [MSH+06a], NZTM [TWGM07], and MM-STM [MM08] use more complex metadata to achieve obstruction freedom.

While inevitability fundamentally introduces blocking into the STM, we see no other obstacles to the use of any of our mechanisms in orec-based STM implementations. In our formulation, inevitability entails only two requirements, the first of which is trivial:

- At most one inevitable transaction: In restricted cases, static analysis may determine that two inevitable transactions will never conflict with one another. It may then be possible to allow multiple concurrent inevitable transactions. However, in general it is unsafe to permit two transactions to run inevitably at the same time, since they may conflict or require exclusive access to the same non-transactional resource. The library exposes a `become_inevitable()` call to transition transactions into inevitable mode. The implementation ensures that at any moment in time, at most one active transaction has returned from a call to `become_inevitable()`. Our implementation of `become_inevitable()` employs a test-and-set lock with exponential backoff to block other inevitable transactions.

- Inevitable transactions do not abort: A transaction that has returned from a call to `become_inevitable()` must not abort if it has performed irreversible opera-
tions. In our mechanisms, writes to shared variables performed by the inevitable transaction are irreversible.

This second requirement is the principal implementation challenge. We further classify the sources of aborts as self, explicit, and implicit, corresponding to aborts caused by program logic, read/write-after-write conflicts, and write-after-read conflicts.

**Preventing Self Abort**  Many STM APIs include a \texttt{retry()} mechanism [HMPH05] for efficient condition synchronization. Clearly this mechanism cannot be used after performing an irreversible operation. It may be possible, however, to perform condition synchronization \textit{before} becoming inevitable or, in certain cases, in ways that are guaranteed never to force rollback of work already performed; we consider these options further in Section 6.4.3.

**Preventing Explicit Aborts**  In all of the orec-based STM algorithms named above, an in-flight transaction must, at some point prior to committing, acquire exclusive ownership of each location it wishes to modify. Subsequent to this acquisition step, any concurrent reader or writer may use contention management [GHP05b; SS05a] to receive permission to abort the exclusive owner. For inevitable transactions that use encounter-time locking, it is straightforward to augment contention management so that non-inevitable transactions always defer to inevitable transactions. This sacrifices nonblocking guarantees, but is otherwise a trivial extension. It also decreases overhead within inevitable transactions, since data written (or overwritten) by inevitable writes need not be logged. Furthermore, given the condition that there is at most one inevitable transaction, this does not introduce deadlock.
**Preventing Implicit Aborts**  While some STM algorithms support a visible reader mode, in which transactions explicitly mark the metadata of locations they read, it is more common simply to record transactional reads in a private log, which is then validated at commit time to make sure that corresponding metadata have not changed. The runtime must ensure that inevitable transactions do not detect such changes after becoming inevitable, via some additional guard on reads. This requirement places the inevitable transaction somewhere between fully “visible” and fully “invisible,” depending on the inevitability implementation.

Implicit aborts can be prevented using a global read/write lock (GRL): a call to `become_inevitable()` blocks until all in-flight transactions commit or abort and clean up metadata. The inevitable transaction then executes, while all other threads block at the start of their next transaction until the inevitable transaction commits. Unfortunately, this solution affords no concurrency, even in workloads with no conflicts. We now turn our consideration to mechanisms that prevent implicit aborts without sacrificing all concurrency.

### 6.3 Inevitability Mechanisms

As suggested in Section 6.2, the primary requirement of a concurrency-permitting inevitability mechanism is a method to guard the inevitable transaction’s read set, so that a conflicting (non-inevitable) writer does not invalidate an inevitable read. In practice, this requirement entails a tradeoff between precision and overhead: more precise mechanisms afford greater concurrency between the inevitable transaction and transactions that write to disjoint areas of memory, but at the cost of higher overhead within all in-flight transactions. We have already described the global read/write lock (GRL).
In the remainder of this section we describe five additional mechanisms that afford progressively higher levels of concurrency.

### 6.3.1 Global Write Lock

Read-only transactions may progress (and commit) alongside an inevitable transaction so long as the inevitable transaction updates the metadata of locations that it writes (thereby enabling the concurrent readers to detect conflicts). Similarly, a writing transaction that uses commit-time locking can progress up to its commit point while an inevitable transaction is in-flight, but must wait to commit until after the inevitable transaction finishes. We refer to this inevitability mechanism as a global write lock (GWL).

When a transaction attempts to become inevitable, GWL makes no assumptions about the state of other threads. The inevitable transaction must thus check metadata before performing any transactional read (requiring a read-before-read (RBR) memory fence on processors with relaxed memory consistency models) and might need to block until a lock is released. To avoid races with non-inevitable writers that have committed or aborted but not yet cleaned up, the inevitable transaction must also use atomic instructions (`compare-and-swap (CAS)` or `load linked/store conditional (LL/SC)`) to update metadata for writes. At the same time, it can elide all read-set logging, and need not postvalidate metadata after a transactional read.

Since inevitable transactions do not validate their read set, a non-inevitable writer must check the GWL flag after acquiring any location, to detect the presence of a concurrent inevitable transaction; if none is detected, the transaction may proceed. Otherwise it must release the location and restart. Testing the GWL flag after acquiring requires a read-before-read/write fence. On architectures like the x86 and SPARC, an
atomic `CAS` provides the required ordering implicitly. On the POWER architecture, a lightweight ISYNC memory fence can be used after the acquires to impose read-before-read/write ordering (which is needed in any case in STM implementations with deferred validation).

GWL allows concurrent non-conflicting read-only non-inevitable transactions to complete during an inevitable transaction’s execution, but we expect GWL to afford only limited parallelism with read-write transactions, since it prevents a non-inevitable writer from committing even when it could logically serialize prior to a concurrent inevitable transaction. We also expect that supporting GWL introduces minimal overhead: with commit-time locking, we add only a single memory fence and comparison to the commit sequence of the non-inevitable transaction. Furthermore, we anticipate that GWL transactions should run significantly faster than their non-inevitable counterparts, even when the semantics of inevitability are not strictly required: there is no need to log and validate the consistency of reads, which usually account for a significant portion of the overhead of non-inevitable transactions.

In GWL, while inevitable reads avoid logging and validation, still each inevitable read must check whether the location to be read is in the process of being written by a concurrent non-inevitable transaction, and if necessary wait until the location is released. In the next two designs, we explore the possibility of eliminating per-inevitable-read overheads completely.

### 6.3.2 GWL + Transactional Fence

Several STM implementations include quiescence mechanisms that permit threads to wait for concurrent transactions to reach predictable points in their execution [DSS06; Fra03; FH07; HSATH06; SMDS07; WCW\textsuperscript{+07}]. These mechanisms support memory
reclamation and privatization, or ensure correctness during validation. By using qui-
escence to eliminate the possibility of not-yet-cleaned-up transactions, we can avoid
the need to instrument inevitable reads, at the cost of a possible delay after becoming
inevitable. (Transactional writes still require instrumentation to ensure that locations
are acquired before they are written. Without this instrumentation, concurrent readers
would not be able to correctly detect conflicts.)

To evaluate the utility of quiescence, we use a Transactional Fence [SMDS07],
which waits for all active transactions to commit or abort and clean up.¹ Once the
transactional fence completes, the inevitable transaction is guaranteed that there will be
no concurrent updates by non-inevitable transactions during its execution.

This same mechanism could be used in conjunction with GRL: upon becoming in-
evitable, a transaction could wait for all concurrent transactions to complete. Since all
new transactions (even those that do not perform writes) would block at their begin
point, once the transactional fence completed, the inevitable transaction would be guar-
anteed exclusive access to shared locations, and could safely perform uninstrumented
reads and writes.

### 6.3.3 Writer Drain

While waiting for all active transactions to finish is a straightforward way to ensure
that an inevitable transaction can always safely read, it is more conservative than nec-
essary. All that is really required is to wait until all concurrent writers have completed.

We can do this by merging a writer count into the GWL flag. We call the resulting

¹Less expensive quiescence mechanisms may also suffice, though they may be specific to particular
TM implementations.
mechanism a **Writer Drain**. With the Drain in place, an inevitable transaction requires neither instrumentation of reads nor atomic instructions in metadata updates for writes.

Abstractly, the Drain is an implementation of GWL in which an inevitable transaction cannot start until all non-inevitable writers have released their metadata. When commit-time locking is used, this behavior can be ensured using a single-word status variable updated according to the protocol shown in Figure 6.1. The status variable consists of three fields: a bit \( i \) indicating an active inevitable transaction, a reservation bit \( r \) indicating a transaction that will become inevitable as soon as extant non-inevitable writer transactions finish, and a count \( n \) of the number of such writers. This protocol resembles a traditional fair reader-writer lock in which inevitable transactions take the role of “writer” and non-inevitable writer transactions take the role of “reader”.

In the **draining** state, thread \( I \) has reserved permission to execute inevitably, but is waiting for all non-inevitable transactions that are about to acquire a location to complete. When the last of these transactions (indicated by an \( n' \) transition) completes,
the drain transitions through a transient state to the **plugged** state. In the **plugged** or **filling** state, thread 1 may execute without restriction, but non-inevitable transactions must block at the point where they wish to acquire a location. If there are pending transactions in the drain when 1 completes, the drain moves to an **unheld** state, in which non-inevitable transactions must inform the drain of their status but may otherwise execute without restriction. When there are no inevitable transactions, and no non-inevitable transactions have acquired locations, the drain is **empty**.

When the drain is **plugged** or **filling**, the inevitable transaction is guaranteed that all metadata is unacquired; it will never encounter a locked location. During its execution, it is also guaranteed that no other transaction will modify metadata. Given these guarantees, the inevitable transaction requires no instrumentation of reads. In addition, though it must still modify metadata before writing the associated data, it can use ordinary writes to do so. In most architectures, such write-before-write ordering has no overhead; in the POWER architecture, it can be achieved with an LWSYNC memory fence.

Like GWL, Drain should only afford limited concurrency with read-write transactions since non-inevitable writers cannot commit during the execution of any inevitable transaction. Furthermore, in the absence of inevitable transactions we expect higher overhead, since writing transactions must increment and decrement the non-inevitable active count in the drain at the beginning and end of their commit sequence (or, in the case of encounter-time locking, before first acquire and after commit, respectively). In the face of any concurrency, these updates will result in adverse cache behavior. However, the throughput of the inevitable transaction should be substantially higher. Consequently, an intelligent just-in-time compiler could reduce the code of an inevitable transaction to almost the size and complexity of a nontransactional equivalent.
6.3.4 Inevitable Read Locks

Both GWL and Drain use a single guard to protect all locations read by the inevitable transaction. Though simple, these mechanisms limit concurrency, since concurrent writers are prohibited from committing even when their write sets do not overlap with the inevitable transaction’s read set. We now consider a precise mechanism to guard only the locations read by the inevitable transaction, thereby allowing a maximum amount of concurrency between the inevitable transaction and disjoint writers.

Our Inevitable Read Locks (IRL) mechanism adds read locks to the underlying STM’s metadata. Typically, lock-based STM overloads a single word with lock and ownership information; the least significant bit is used to indicate locking, with all other bits forming a version number or pointer to the lock holder. Assuming that all pointers must be aligned to at least 4 bytes, we can take another low-order bit and use it to indicate that the location is being read (but not written) by the inevitable transaction.\(^2\)

Using IRL, inevitable transactions must read and (potentially) atomically update metadata on every transactional read, and must release read locks (using normal stores) upon commit. During a read or write, the inevitable transaction may detect a conflict with an in-flight transaction writing the same location. To resolve the conflict, the inevitable transaction must issue a remote abort, perhaps after briefly waiting for the conflicting transaction to complete. In STM implementations that use encounter-time locking with in-place update, the inevitable transaction may need to wait after issuing a remote abort, so that the aborted transaction can undo any modifications it made.

Non-inevitable transactions also require minor modification to support IRL. During reads, transactions may ignore benign conflicts with an inevitable reader (detected when the transaction reads an ownership record and finds the read bit set), but during

\(^2\)This mechanism was also proposed by Welc et al. [WSAT08].
transactional writes, the runtime must prevent acquisition of locations that are actively being read by an inevitable transaction. To resolve such conflicts, the writer can either block or abort itself.

While IRL should provide a high degree of concurrency between inevitable transactions and nonconflicting writing transactions, we expect performance to be limited for a number of reasons. First, inevitable transactions incur significant overhead due to a high number of expensive atomic operations. Second, inevitable transactions must still bookkeep their reads, so that they can release their read locks upon commit. Worse yet, the additional bus messages generated by the increased number of atomic operations may cause a slowdown for truly disjoint non-inevitable transactions. Lastly, as with visible reads in STM, we expect decreased scalability due to cache effects: updates to metadata by the inevitable transaction cause additional bus messages and cause cache misses when concurrent non-inevitable readers perform validation.

### 6.3.5 Inevitable Read Filter

By approximating the set of read locks as a Bloom filter [Blo70], our Inevitable Read Filter (Filter) mechanism decreases impact on concurrent reads, albeit at the expense of a more complex protocol to handle non-inevitable writes and inevitable reads. An inevitable transaction records the locations it reads (or the locations of the associated metadata) in a single, global Bloom filter, and writing transactions refrain from acquiring locations recorded in the filter. The size of the filter and the set of hash functions are orthogonal to the correctness of the mechanism, and serve only to decrease the frequency of false conflicts.

Since inevitable readers and non-inevitable writers both must interact with the filter and with ownership metadata, we must take care to avoid data races. The non-inevitable
writer always acquires a location before checking the filter, and upon a positive lookup in the filter the writer must abort. Similarly, the inevitable reader always records locations in the filter before checking metadata, and if that subsequent metadata check fails, the inevitable transaction blocks until the ownership record is released.

The complexity of the above protocol to interact with the filter and ownership metadata is compounded by the processor memory model; the inevitable transaction must write to the filter before reading a location, and non-inevitable transactions must acquire locations (via atomic writes) before testing whether those locations are present in the filter. In the former case, an explicit write-before-read (WBR) memory fence is required on the x86, SPARC, and POWER architectures. The latter case introduces the need for a full sync instruction (SYNC) on the POWER, but no additional ordering on x86 or SPARC. With commit-time locking, a single SYNC suffices for the non-inevitable transaction, as it can acquire all needed metadata, issue the SYNC, and then test all acquired locations in the filter. Until hardware designers decrease the best-case overhead of WBR fences, their presence on the critical path of committing transactions introduces significant overhead.

Due to the unavoidable requirement for WBR ordering, we expect high overhead for the inevitable transaction using Filter. However, we expect better scaling than IRL, since read-read concurrency does not introduce any overhead, and does not sacrifice the read-write concurrency of IRL. The ability to tune the filter by changing its size or hash functions may prove useful as developers gain experience with transactional workloads.
Mechanism  | Delay upon becoming inevitable | concurrent read-only | concurrent writes |
--- | --- | --- | --- |
GRL  | Yes | No | No |
GWL  | No | Yes | No |
GWL + Fence  | Yes | Yes | No |
Drain  | Sometimes | Yes | No |
IRL  | No | Yes | Yes |
Bloom  | No | Yes | Yes |

Table 6.1: Summary of benefits and drawbacks of different inevitability options.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Inev Read Instr</th>
<th>Inev Write Instr</th>
<th>Inev Read Logging</th>
<th>Inev Commit Overhead</th>
<th>Begin Overhead</th>
<th>NonInev Commit Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRL</td>
<td>None</td>
<td>None</td>
<td>No</td>
<td>None</td>
<td>WBR</td>
<td>N/A</td>
</tr>
<tr>
<td>GWL</td>
<td>Wait</td>
<td>Acquire</td>
<td>No</td>
<td>None</td>
<td>None</td>
<td>Test</td>
</tr>
<tr>
<td>GWL + Fence</td>
<td>None</td>
<td>Store</td>
<td>No</td>
<td>None</td>
<td>WBR</td>
<td>Test</td>
</tr>
<tr>
<td>Drain</td>
<td>None</td>
<td>Store</td>
<td>No</td>
<td>CAS</td>
<td>None</td>
<td>2 CAS</td>
</tr>
<tr>
<td>IRL</td>
<td>Acquire</td>
<td>Acquire</td>
<td>Locks</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Bloom</td>
<td>Write</td>
<td>Acquire</td>
<td>Filter</td>
<td>None</td>
<td>None</td>
<td>WBR Intersect</td>
</tr>
</tbody>
</table>

Table 6.2: Overheads imposed by inevitability mechanisms.

### 6.3.6 Summary

Table 6.1 summarizes the impact of each mechanism on the behavior of concurrent non-inevitable transactions, and also identifies which mechanisms may introduce a delay in the inevitable transaction at its begin point. In Section 6.4 we discuss limitations that an inevitability mechanism imposes on the sorts of irreversible operations that can be performed; those limitations may, in turn, require a fallback mechanism to use GRL when the properties of another inevitability mechanism prove inadequate. Table 6.2 summarizes the sources of latency introduced by each mechanism.
6.4 Programming with Inevitability

We now assess the impact of inevitability on transactional programming models. For simplicity and generality, we assume that the STM is implemented as a library, and we ignore the concurrent execution of transactional and nontransactional code.

6.4.1 Irreversible Operations

In this section we contrast inevitability with other ways of accommodating irreversible operations, centering the discussion on the desirable properties that are intrinsic to inevitability.

Enforcing Mutual Exclusion of Multi-Instruction I/O

Alternatives to inevitability for I/O include buffering, open nested transactions [NMAT+07], and punctuating transactions [SKBY07]. Buffering is not a general solution for interactive I/O. We therefore limit our discussion to the other two alternatives. For each of these, we consider their use in a simple, output-only operation.

Listing 6.1 presents transactional operations for an abstract set initialized via the `init()` function. We consider an array of such sets, accessed via `WorkLoad1` from Listing 6.2. For \( N > 1 \) threads, each thread \( n \in N \) executes `WorkLoad1(n)`. We assume that `SetPrint()` is executed inevitably, via an API call that correctly ensures at most one inevitable transaction in flight at any time.

Since inevitability enforces mutual exclusion, there will be no interleaving of print operations among threads. Thus while the order in which \( N \) sets are printed cannot be predicted, inevitability guarantees that each of the \( N \) sets will be printed without interruption by concurrent printing threads. This property does not hold for open nesting or punctuating transactions.
Set sets[numthreads]

init:
  1  foreach(set in sets)
  2    set.fill_with_random_vals()

SetPrint(Set S):
  1  atomic
  2   print("Set " & S.id)
  3   foreach (s in S)
  4     print("  " & s.toString())
  5   print("End of set " & S.id)

SetAdd(Set S, value v):
  1  atomic
  2   if (!S.contains(v))
  3     S.add(v)

SetRemove(Set S, value v):
  1  atomic
  2   if (S.contains(v))
  3     S.remove(v)

Listing 6.1: Add, Remove, and Print transactions for a simple set.

Under open nesting, calls to SetPrint can abort; consequently, the programmer must specify compensating actions to undo or mitigate the effects of print instructions issued from transactions that call SetPrint but do not commit. Since such actions are properties of the specific irreversible operation performed, it appears unlikely that a single design pattern or library call can be provided to make this task easy for programmers.

With punctuated transactions, individual I/O operations need not be rolled back. However, at each I/O point, the transaction effectively commits and starts a new transaction; to ensure correctness, the programmer attaches a set of predicates to the punctuating action (in this case, the call to print), specifying what conditions must be restored before the thread resumes a new transactional context that appears to be a con-
WorkLoad1(int tid):
1   SetPrint(sets[tid]);

WorkLoad2(int tid):
1   do
2       if (tid == 0)
3           SetPrint(sets[0])
4       else if (tid % 2 == 0)
5           SetAdd(sets[0], random())
6       else
7           SetRemove(sets[0], random())
8   until (TimerInterrupt)

Listing 6.2: Parallel workloads for an array of sets.

continuation of its previous context. Unfortunately, at the point where the transaction is punctuated, the scheduler may permit any concurrent thread to execute the first of its print statements. We believe that this problem can be prevented through the explicit creation of a monitor-like mechanism that uses additional punctuating transactions to synchronize calls to SetPrint. However, even if the runtime provides such a mechanism, it would introduce a risk of deadlock and would reintroduce the need for global reasoning about locks.

Preserving Local Reasoning About Correctness While inevitability is a global property (at most one active transaction can be inevitable), it does not violate abstraction boundaries or require global reasoning about correctness. WorkLoad2 from Listing 6.2 demonstrates how alternatives to inevitability such as privatization and punctuating transactions can necessitate deep changes to the structure of an algorithm.

Under WorkLoad2, a single thread attempts to print a set that is being actively modified. The runtime must ensure that the output matches some dynamic instance of the set. Let us suppose that the set initially contains all odd values within the range
after thread0 prints values \( n \ldots m/2 \), the elements \( n \ldots m \) are all removed from the set, and then all even elements in the range \( m/2 \ldots m \) are added. With punctuating transactions, all set modifications can complete during a break in the execution of thread0. If thread0 continues printing, the output will not match any dynamic instance of the set. However, encoding an invariant to capture the requirement that all previously printed values remain in the set may be unacceptably complex: it must at least assert that all printed values are still in the set, and may also need to understand the set iterator’s implementation, so it may assert that there have not been updates that fall within the range already printed.

The set can be printed correctly if privatization is used to lock the set, via the flag-based privatization idiom [MBS+08b]. However, privatization precludes concurrent accesses that do not conflict with the I/O, such as updates to the portion of the set that has not yet been printed. Additionally, flag-based privatization requires that every operation that might update the data structure reads the flag, which adds overhead even when I/O is rare. To correctly publish the data structure after I/O completes, flag-based privatization also requires either limits on compiler reorderings of transactional accesses, or else overhead on transaction boundaries [MBS+08a]. Furthermore, concurrent accesses that encounter a privatized set must abort, necessitating support for \texttt{retry()} by the runtime.

### 6.4.2 Non-Transactional Code and System Calls

A limitation of library-based STM is its inability to call precompiled code. Recompilation of libraries can solve this problem, as can dynamic binary rewriting [FFM+07; OCS07; YWWJ06]. Without such mechanisms, precompiled libraries and system calls
can still be called safely from an inevitable transaction, so long as the inevitability mechanism and the library code obey the following rules.

First, the underlying STM must not use indirection. In an indirection-based STM algorithm [Fra03; FH07; HLMS03; MSH+06a; MSS05], read instrumentation is necessary in inevitable transactions, because data does not reside in place. Indirection-free systems have no such constraint, because we use encounter time locking and in-place update for the inevitable transaction.

Second, if the library code reads and writes shared locations, it must be instrumented (though not necessarily by the library itself) according to the inevitability mechanism. We provide inevitable prefetch instructions for this purpose: a write prefetch locks its corresponding location without modifying the value, and a read prefetch performs the appropriate read instrumentation (such as adding locations to a filter, acquiring read locks, or ensuring that write locks are not held). Without prefetching, only GRL can call precompiled code that writes locations that have not been previously written by the calling transaction, and GWL, IRL, and Filter cannot call precompiled code that reads new locations. With prefetching, library code can be called only if its writes can be conservatively predicted. Furthermore, if the underlying inevitability mechanism is not Drain or GWL+Filter, the code’s reads must also be predictable. In summary, inevitability and prefetching together can be used to call some, but not all, precompiled library code.

Using inevitability to call precompiled code decreases scalability relative to dynamic binary rewriting and recompilation. Inevitability serializes transactions even when the library code they execute does not perform irreversible operations. Furthermore, some rewriting mechanisms achieve the effect of speculative lock elision when calling lock-based code [OCS07]. In contrast, inevitability ensures that the calling
transaction will not abort while holding a lock, (avoiding the chance of deadlock if the lock is not released correctly), but cannot extract parallelism that may be available.

### 6.4.3 Condition Synchronization

Whereas inevitability improves transactional programming by adding support for irreversible operations, system calls, and precompiled libraries, it complicates the use of condition synchronization via explicit self-abort. As mentioned in Section 6.2, we explicitly forbid inevitable transactions from self-aborting after performing irreversible operations.

This condition does not, however, prevent all condition synchronization within inevitable transactions. As in X10 [CDE+05], an inevitable transaction can certainly synchronize at its begin point, though this form of condition synchronization is not compatible with nesting. More generally, an inevitable transaction can synchronize before executing `become_inevitable`, and at any point in its inevitable execution at which it has only performed reads (writes, library calls, and system calls must be conservatively considered irreversible operations).

Additionally, an inevitable transaction can synchronize using self-abort after performing irreversible operations in systems that provide closed-nested transactions, as long as (1) the retry is performed within a specially annotated closed-nested transaction; (2) the retry condition involves only reads to locations that were not accessed inevitably; (3) the inevitability mechanism permits concurrent writers to commit; and (4) the condition can be computed and written by any transaction without conflicting with the inevitable transaction’s reads and writes. Since these properties do not cleanly compose and cannot in general be statically checked, we expect the usefulness of such condition synchronization to be limited.
6.5 Evaluation

In this section we analyze the performance of our different inevitability mechanisms across several microbenchmarks. We conducted all tests on an IBM pSeries 690 (Regatta) multiprocessor with 32 1.3 GHz Power4 processors running AIX 5.1. Our STM runtime library is written in C and compiled with gcc v4.0.0. All benchmarks are written in C++ and compiled using g++ v4.0.0. Each data point represents the average of five trials, each of which was run for five seconds.

STM Library Configuration Our STM library is patterned after the per-stripe variant of TL2 [DSS06], and uses commit-time locking with buffered updates. We use an array of 1M ownership records, and resolve conflicts using a simple, blocking contention management policy (abort on conflict).

Inevitability Model Due to the lack of standard transactional benchmarks, much less those requiring inevitability, we use parameterized microbenchmarks to assess the impact on latency and scalability imposed by our algorithms. For each test, all threads are assigned tasks from a homogeneous workload. A single thread is instructed to perform each of its transactions inevitably, with all other threads executing normally. In this manner, we can observe the impact on scalability introduced by frequent, short-lived inevitable transactions, as well as the effectiveness of different inevitability mechanisms in speeding a workload of non-synchronizing transactions. Our choice to execute all inevitable transactions from the same thread increases the predictability and regularity of our results by avoiding cache contention for the inevitability token.

Our decision to analyze short inevitable transactions is deliberate. Baugh and Zilles [BZ07] have argued that transactions that perform I/O are likely to run a long
time, and to conflict with almost any concurrent activity. This suggests that quiescence overhead will be an unimportant fraction of run time, and that there is little motivation to let anything run in parallel with an inevitable transaction. While this may be true of operations that write through to stable storage, it is not true of more lightweight kernel calls, or of calls to pre-existing libraries (including buffered I/O) that are outside the control of the TM system. As an example use case, we have developed a graphical game, “swarm,” that uses transactions to synchronize with concurrent 3-D rendering in OpenGL [SSD+08]. Without inevitability, the rendering thread may be forced to perform excessive copying (application-level buffering), or else to render a single visual object over the course of many separate transactions, at the risk of inconsistent output.

**Inevitability Mechanisms** For each workload, we evaluate 9 library variants. The default (Baseline) variant is our standard STM implementation with no inevitability support. For programs with inevitable transactions its behavior is incorrect, but useful for comparison purposes. We compare against the global read lock (GRL); the global write lock both with and without a Transactional fence (GWL+Fence and GWL, respectively); the writer drain (Drain); inevitable read locks (IRL); and three Bloom filter mechanisms that differ in the size of the filter used and the number of hash functions. Filter (s) uses a single hash function and a 64-bit bloom filter. Filter (m) and Filter (l) both use a 4096-bit filter, with one and three hash functions, respectively.

### 6.5.1 Latency for Non-Inevitable Transactions

Our various inevitability algorithms differ in the amount of overhead they place on the critical path of non-inevitable transactions. We first consider the case when inevitability is not used by any transaction. Figure 6.2(a) compares overheads on a
microbenchmark in which threads access disjoint regions of memory. Transactions are read-only with 33% probability, and otherwise perform 20% writes. Each writing transaction accesses 20 distinct locations (thus there are 4 writes), but the locations within each thread’s disjoint memory region vary.

Our hope for linear scaling is not realized, due to inherent serialization among writing transactions in the STM runtime. However, Drain serializes much earlier, due to contention for its status variable. The two atomic operations required to enter and exit the drain cause substantial bus traffic and cache misses; at any significant level of concurrency, each of these operations will result in a miss. Excluding Drain, our mechanisms introduce only modest overheads; they are all within 10% of baseline performance. This behavior matches our expectations, and confirms that supporting inevitability need not, in and of itself, be a significant source of latency.

### 6.5.2 Supporting Disjoint Writes

When all transactions’ writes sets are disjoint, inevitability should ideally have no impact on scalability. In Figure 6.2(b), we show a benchmark in which every thread accesses 100 locations per transaction. Again, 33% of transactions are read-only, with the remaining transactions performing a mix of 20% writes to private buffers and 80% reads. However, all reads are to a single shared structure. Given the size of each transaction, we do not expect TL2’s global timestamp to be a bottleneck, although we do expect a bottleneck in Drain.

Naturally, we expect linear scaling from the Baseline implementation, and we might hope that inevitability would speed up at least the inevitable thread without sacrificing scalability. However, our mechanisms incur various penalties that prevent this hope from being realized. With such large read sets, the absence of read instrumentation for
(a) Disjoint Transactions, 20 accesses per transaction, 20% writes, using a TL2-like runtime. Inevitability is not used by any thread.

(b) Shared reads, disjoint writes, 100 accesses per transaction, 20% writes, using a TL2-like runtime. Thread 0 runs inevitably at all times.

Figure 6.2: Inevitability Workloads
Drain, GRL, and GWL+Fence results in substantial single-thread speedup. However, GWL forbids concurrent write commits entirely, resulting in flat performance. When the Transactional Fence is added to GWL, the periods when the inevitable transaction is blocked provide an opportunity for concurrent non-inevitable transactions to commit, raising performance. Furthermore, the GWL+Fence outperforms GRL by allowing concurrent transactions to progress up to their commit point during the inevitable transaction’s execution.

IRL and the large and medium Filters perform slightly worse than Baseline, due to their additional memory ordering constraints. The workload is clearly sensitive to Filter parameters: the small Filter causes unnecessary aborts and performs dramatically worse than the other two. The distance between Baseline and these scalable mechanisms increases slightly as concurrency increases: we attribute this to the increased cache misses that result from read locks covering shared locations, and from cache misses during Filter accesses by non-inevitable transactions.

When we simulated the WBR memory fence with a lightweight-sync instruction, performance of the inevitable thread improved by 60% for the large and 53% for the small and medium Filters. If we take this to be a reasonable best-case overhead for a WBR fence, then the Filters should all outperform Baseline at one thread. The impact on non-inevitable transactions is negligible (less than 1% on average) since they issue only one WBR per writing transaction.

In summary, the mechanisms we expected to scale do so, though not quite as well as Baseline. Drain consistently outperforms quiescence, and affords better scalability than GWL, because it blocks non-inevitable transactions before they acquire locations. In contrast, GWL often detects conflicts between non-inevitable and inevitable transac-
tions after acquisition, which forces the non-inevitable transaction to abort in order to prevent deadlock.

### 6.5.3 Workload Acceleration

We lastly consider the effectiveness of inevitability as a performance optimization. In the large body of worklist-style algorithms that have occasional conflicts, we posit that inevitability can improve performance. In particular, when there is no priority or fairness requirement among tasks, and when tasks do not synchronize with each other, then the decision to execute some tasks inevitably can improve throughput, so long as inevitable transactions execute more quickly than their non-inevitable counterparts. To evaluate this claim, we concurrently execute an equal mix of insert, remove, and lookup instructions on a set, and measure the change in throughput when one thread executes all transactions inevitably. Figure 6.3 presents two such workloads. In the first, tasks access a 256-element hash table; in the second, a 1M-element red-black tree.

The HashTable’s small transactions do not benefit from inevitability; the overhead of the drain and the TL2 timestamp dominate, resulting in both a limit on the scalability of an otherwise scalable benchmark, and a limit on the improvement afforded by inevitability. In the RBTree, however, transactions are large enough that the drain overhead does not dominate. Consequently, there is a modest but decreasing benefit to inevitability at lower thread levels, and not until 24 threads does inevitability increase latency.

Again, the GWL performs worse than GRL. As before, most of GRL’s scalability is due to non-inevitable transactions completing while the inevitable transaction is blocked. In GWL, however, non-inevitable transactions can slow down an inevitable transaction, since they access metadata concurrently with the inevitable trans-
Figure 6.3: Scalable worklists, using a TL2-like runtime. Thread 0 runs inevitably at all times. Transactions in the top graph are short, modeled with a 256-element hash table, while transactions in the bottom figure are larger, via a red-black tree with 1M elements.
action. Since all non-inevitable transactions block in GRL, no such interference occurs. Adding the Transactional Fence to GWL eliminates this effect. However, as in GRL, the fence prevents fast-running inevitable transactions from running often enough to improve performance at high thread counts.

In additional experiments, we confirmed that this effect is even more pronounced in workloads with less inherent concurrency—e.g., with large transactions that frequently conflict. Here single-thread performance is paramount, and the use of inevitability to raise that performance can have a dramatic impact. On a RandomGraph [MSS05] workload, where all transactions conflict and each transaction reads hundreds of locations, we observed over two orders of magnitude improvement at one thread; at higher thread levels, contention for the drain increased, eventually reducing performance to baseline levels at 16 threads. In a scalable LinkedList workload with large read sets, inevitability raised throughput on 1–32 threads to a constant level 50% higher than the peak performance without inevitability.

### 6.5.4 Dependence on Workload Characteristics

Our results suggest that the best inevitability mechanism depends on the offered workload:

**Library or system calls with unpredictable write sets:** GRL is the only option in this case. It sacrifices most concurrency in the application.

**Short inevitable transactions that are likely to conflict with non-inevitable transactions:** Here GWL is attractive. It requires, however, that the read and write sets
of precompiled functions be predicted, and it limits scalability if there are concurrent nonconflicting writers.

**Long but rare inevitable transactions that call library code with unpredictable read sets:** GWL+Fence should perform well in this case. The cost of the fence is offset by the improved performance of the inevitable transaction, and when the workload does not contain concurrent disjoint writers, the impact on scalability will be limited.

**Long, frequent inevitable transactions that run with long non-inevitable transactions, and rarely conflict with them:** Drain seems best for these. It allows calls to precompiled code with unpredictable read sets, but introduces a scalability bottleneck if there are many short-running nonconflicting writers.

**Short, frequent inevitable transactions that run with short non-inevitable transactions, and rarely conflict with them:** Both IRL and Filter should work well here. Both are well-suited to workloads with frequent inevitable transactions and concurrent, nonconflicting writer transactions. They both afford good scalability, but require that the read and write sets of library code be predicted. The choice of mechanism should depend on the timing of the call to `begin_inevitable`. If the call is made late, the expense of making existing reads inevitable should be lower for Filter, since the cost of a single memory fence can be amortized across all read locations, and there are no atomic operations. Filter should also be preferred if the inevitable transaction’s read set overlaps with other transactions’ reads, since IRL’s read locks will cause non-inevitable transactions to incur cache misses.

Ultimately, we expect that a combination of programmer input, static analysis, and run-time profiling will be necessary to find the best mechanisms for a particular work-
load. Clearly, some mechanisms are unsuitable for certain I/O operations and system calls. Additionally, the best choice of mechanism is likely to depend on machine characteristics (e.g., the latency of WBR fences, \texttt{CAS} instructions, and cache misses), the amount of parallelism (especially if the Drain mechanism is an option), and the frequency and duration of inevitable transactions.

### 6.6 Conclusions

In order for transactions to replace locks, the underlying TM implementation must let programmers perform I/O and irreversible operations from within transactions. Additionally, the requirement to do so should not interfere with either the latency of transactions (regardless of whether the mechanism is used or not), or the overall throughput of the system. In this chapter we presented several mechanisms to implement inevitable transactions while attempting to comply with these goals.

Using our mechanisms, programmers can easily incorporate I/O, system calls, and other irreversible operations into STM-based applications. While no single mechanism achieves both low contention-free overhead and high scalability at high levels of concurrency, these mechanisms make it practical to develop realistic transactional workloads. In particular, inevitability provides a simple and effective way to allow I/O in transactions and, contrary to conventional wisdom, need not sacrifice scalability for workloads that have few write conflicts between inevitable and non-inevitable transactions.

The Bloom Filter mechanism underscores the importance of designing memory systems with lower write-before-read fence overhead. Even with current overheads, the Filter seems like best way to provide inevitability without sacrificing scalability. When
speed within inevitable transactions is paramount, the Drain appears to be best: it gives high single-thread performance and low overhead when conflicts are infrequent, but introduces a scalability bottleneck. We suspect that it is most suitable as a performance optimization for workloads with low contention and no condition synchronization. Alternatively, GWL+Fence provides better scalability and comparable throughput for inevitable transactions with little impact on non-inevitable transactions, at the cost of delays at the point when a transaction becomes inevitable. Ideally, future STM systems might employ static analysis or profiling to identify the ideal inevitability mechanism for a given application.
7 Ordering-Based Semantics for Software Transactional Memory

One of the goals of this dissertation is to identify orthogonal mechanisms that improve the generality of STM. To this end, Chapter 5 presented an orthogonal mechanism for programmer-defined priority, and Chapter 6 presented a number of mechanisms that allow transactions to perform I/O. In this chapter, we consider interactions between transactions and nontransactional code. Our focus is weakly isolated STM, in which races between transactions and nontransactional code are considered program bugs. The central challenge, then, is ensuring that the STM runtime does not introduce artificial races when data is transitioned between shared and private states.

Our presentation of the global commit counter (Chapter 2) and RingSTM (Chapter 3) included mechanisms that addressed specific elements of a language-level semantics (the problems of doomed transactions and delayed cleanup, as described in Section 1.5.3). In this chapter we present a semantics first, and then discuss implementation. Furthermore, since we envision TM as means to save programmers from the need to reason about locks, we will build our semantics around a global total order for transactions, rather than single global lock atomicity. We argue that ordering-based semantics are conceptually preferable to lock-based semantics, and just as efficient.
7.1 Introduction

Transactions constrain the ways in which thread histories may legally appear to interleave. If all shared data were accessed only within transactions, the constraints would be relatively simple. In a break from the database world, however, memory transactions are generally expected to coexist with nontransactional memory accesses, some of which may also access shared memory. This coexistence complicates the task of specifying, for every read in the program, which values may be returned.

Many possible semantics for Transactional Memory (TM) have been proposed, including strong and weak isolation (also known as strong and weak atomicity) [BLM05; SMAT+07], single lock atomicity (SLA) [LR07; MBS+08a; MBS+08b], and approaches based on language memory models [GMP06], linearizability [GK08; Sco06], and operational semantics [ABHI08; MG08]. Of these, SLA has received the most attention. It specifies that transactions behave as if they acquired a single global mutual exclusion lock. Unfortunately, as several groups have noted [LR07; MBS+08a; MSS08; SMDS07; SDMS08], SLA requires behavior that can be expensive to enforce, particularly when a thread privatizes shared data (rendering it logically inaccessible to other threads), works on it for a while (ideally without incurring transactional overheads), and then publishes it again.

Most software TM (STM) implementations today do not correctly accommodate privatization and publication, and forcing them to do so—at the boundaries of every transaction—would impose significant costs. In an attempt to reduce those costs, Menon et al. [MBS+08a] have proposed a series of semantics that relax the requirement for serialization among transactions. These semantics are described in terms of locking protocols significantly more complex than SLA. While we appreciate the motivation, we argue, with Luchangco [Luc08], that locks are the wrong way to formalize
transactions. We prefer the more abstract approach of language-level memory models [BA08; GMP06; MPA05]; like traditional formalizations of database semantics, these directly specify permissible access orderings. We also argue that if one wishes to reduce the cost of SLA, it makes more sense to relax the ordering between nontransactional and transactional accesses within a single thread, rather than the ordering among transactions.

We argue that SLA is equivalent to the traditional database ordering condition known as strict serializability (SS). As a candidate semantics for STM, we suggest transactions with selective strict serializability (SSS), in which nontransactional accesses are ordered with respect to a subset of their thread’s transactions. Whether this subset is all, none, or some explicitly or implicitly identified set in between, a single formal framework suffices to explain program behavior. We also propose a slightly weaker semantics, selective flow serializability (SFS), that orders nontransactional accesses with respect to subsequent transactions in other threads only in the presence of a forward dataflow dependence that could constitute publication.

Like Menon et al., we take the position that races between transactional and nontransactional code are program bugs, but that (as in Java) the behavior of buggy programs should be constrained to avoid “out of thin air” reads. SSS and SFS allow the programmer or compiler to eliminate races by labeling a minimal set of transactions, while still constraining behavior if the labeling is incorrect.

After a more detailed discussion of background in Section 7.2, we formalize our ordering-based semantics in Section 7.3. To the best of our knowledge, this is the first attempt to formalize privatization and publication safety without recourse to locks. We discuss implementation options in Section 7.4, in the context of an STM system
patterned on TL2 [DSS06]. We compare the performance of these implementations experimentally in Section 7.5.

7.2 Background

Arguably the most intuitive semantics for transactions would build on sequential consistency, providing a global total order, consistent with program order, for both transactions and nontransactional accesses. Unfortunately, most architects already consider sequential consistency too expensive to implement, even without transactions. Among other things, it precludes standard compiler optimizations like write reordering. In light of this expense, sequentially consistent transactions appear to be a non-starter.

Several researchers have argued instead for what Blundell et al. [BLM05] call strong atomicity, otherwise known as strong isolation [LR07]. This drops the requirement that a thread’s nontransactional memory accesses be seen by other threads in program order. It still requires, however, that nontransactional accesses serialize with respect to transactions; that is, that each nontransactional reference appears to happen between transactions, and that all threads agree as to which transactions it appears between.

Unfortunately, strong isolation still suffers from several significant costs, which make it unappealing, at least for software implementation:

**Granularity:** In a high-level programming language, it is not immediately clear what constitutes an individual memory access. Is it acceptable, for example, for a transaction to intervene between the read and write that underlie \( x++ \) on a load-store machine? How about \( x = 0x300000003 \), where \( x \) is a `long long` variable but hardware does not provide an atomic 64-bit store?
**Instrumentation:** It is generally agreed that speculative execution of software transactions in the face of concurrent nontransactional accesses will require the latter to inspect and (in the case of writes) modify transactional metadata. Particularly in programs that make extensive nontransactional use of data that are temporarily private, this instrumentation can have a major performance impact. Abadi *et al.* recently developed a technique to avoid nontransactional inspection of metadata [AHM09]. However, this technique operates at a page-level granularity, and can result in substantial overhead if shared and private data reside on the same page.

**Optimization obstruction:** Nontransactional accesses cannot safely be reordered if they refer to locations that may also be accessed by transactions. The sequence
\[
x = 1; \ y = 2
\]
adopts the possibility that a concurrent transaction will see
\[
x == 1 \ \&\& \ y != 2.
\]
If the compiler is unable to prove that no such transaction (or series of transactions) can exist, it must, (a) treat \( x \) and \( y \) as volatile variables, access them in program order, and insert a write-write memory barrier between them; or (b) arrange for the assignments to execute as a single atomic unit.\(^1\)

In light of these costs, most STM systems provide some form of *weak isolation*, in which nontransactional accesses do not serialize with transactions. As several groups have noted [GMP06; MBS’08a; MG08; SMDS07], the exact meaning of weak isolation is open to interpretation. Perhaps the most popular interpretation is **single lock atomicity (SLA)** [LR07, page 20], which states that transactions behave as if they held a global mutex lock. Unfortunately, even these semantics have nontrivial cost, and are

---

\(^1\)Note that while strong isolation is sometimes equated with making every nontransactional access execute as if it were an isolated transaction [BLM05], this characterization is problematic: it could necessitate a global total order not only between these isolated transactions and “real” programmer-specified transactions, but *among* the isolated transactions. The end result would be equivalent to sequential consistency for shared locations.
Listing 7.1: Examples of publication (left) and privatization (right). More obscure examples, involving antidependences or even empty transactions, are also possible [MBS+08a].

unsupported by most TM implementations, particularly for programs that publish or privatize data.

Publication (Listing 7.1, left) occurs when a thread initializes or otherwise modifies a data structure that is logically private, and then modifies shared state to make the structure accessible to other threads. Privatization (right) is the reverse: a thread’s modification of shared state makes some structure logically private. The appeal of privatization and publication is the possibility that temporarily private data might be accessed without transactional overhead.

Traditionally, when a thread releases a mutual exclusion lock, all prior accesses by the thread are guaranteed to have occurred from the point of view of every other thread. Similarly, when a thread acquires a lock, all subsequent accesses by the thread are guaranteed not to have occurred. These facts suggest that the natural implementation of SLA would be publication- and privatization-safe.

The Privatization Problem. In previous work [SMDS07], we identified two dimensions of the privatization problem. Both arise from the fact that STM systems may perform operations that logically precede, but physically follow, a transaction’s linearization point. If nontransactional code is unaware of the behavior of the TM system, transactional and nontransactional work may overlap.
The first “half” of the privatization problem is the delayed cleanup problem (also described by Larus and Rajwar [LR07, pages 22–23]), where transactional writes may appear to occur too late from the point of view of nontransactional code. Specifically, a thread that privatizes shared data may fail to see logically prior updates by a transaction that has committed but has not yet written its “redo log” back to main memory. The second “half” of the privatization problem is the doomed transaction problem (also described by Wang et al. [WCW+07, pages 6–7]), where private writes may appear to occur too early from a doomed transaction’s point of view. The resulting inconsistent view of memory may then allow that transaction to fall into an infinite loop, suffer an exception, or (in the absence of run-time sandboxing) perform erroneous actions that cannot be undone.

The Publication Problem. One might not initially expect a publication problem: private accesses are assumed to occur before publication, and there is no notion of cleanup for nontransactional code. Menon et al. show, however, that problems can arise if the programmer or compiler prefetches data before it is actually published (Listing 7.2).

Relaxing SLA. Under SLA, straightforward solutions to the privatization and publication problems [MBS+08a; SMDS07] require transactions to begin and clean up in serialization order. While heuristic optimizations may relax this requirement in some cases [MSS08], it seems clear that the general case will remain expensive for STM. To reduce this expense, Menon et al. propose to relax the ordering among transactions. Of their three candidate semantics, asymmetric lock atomicity (ALA) seems most likely to reduce transactional overhead without precluding standard compiler optimizations.

2Conversely, in an STM system based on undo logs, a privatizing thread may see erroneous updates made (temporarily) by a transaction that has aborted but not yet cleaned up. As observed by Menon et al., such reads appear to be fundamentally incompatible with the prohibition against “out of thin air” reads. We therefore assume a redo log in the remainder of this chapter.
Listing 7.2: Publication (left) in parallel with a safe (middle) or unsafe (right) use of published data. The programmer has made B a transaction to ensure that it orders, globally, after prior initialization of x. Vertical spacing is meant to suggest a possible interleaving of operations across threads. Both D and F will serialize after B.

ALA transactions behave as if (1) there is a separate reader-writer lock for every datum, (2) read locks are acquired (presciently) at the beginning of the transaction, and (3) write locks are acquired immediately prior to writing. The asymmetry of reads and writes reflects the fact that (a) in most TM systems it is much easier for a reader to detect a conflict with a previous writer than vice versa, and (b) in most programs publication can be assumed to require a write in one transaction followed by a read in another.

In our view, ALA and similar proposals suffer from three important problems. First, they explain transaction behavior in terms of a nontrivial fine-grain locking protocol—something that transactions were intended to eliminate! Second, they give up one of the key contributors to the success of database transactions—namely serializability (see for example Fig. 11 of Menon et al. [MBS+08a]). Third, they impose significant overheads on transactions that do serialize, even in the absence of publication and privatization.
7.3 Ordering-Based TM Semantics

Our proposed alternative to lock-based semantics begins with a programming model in which, in every execution history $H$, each thread $i$ has a memory access history $H^i \subset H$ in which certain maximal contiguous strings of accesses are identified as (outermost) transactions. We use $T_H$ to denote the set of all transactions. We do not consider open nesting here, nor do we permit overlapping but non-nested transactions. Moreover, from the programmer’s point of view, transactions are simply atomic: there is no notion of speculation or of committing and aborting. A typical implementation will need to ensure that abortive attempts to execute a transaction are invisible; among other things, this will require that such attempts retain a consistent view of memory [GK08].

The goal of a semantics for TM is to constrain the ways in which thread histories may legally interleave to create a global history. In keeping with the database literature and with memory models for programming languages like Java [MPA05] and C++ [BA08], we believe that the appropriate way to specify these constraints is not by reduction to locks, but rather by specification of a partial order on program operations that restricts the set of writes that may be “seen” by a given read.

7.3.1 Strict Serializability

The standard database ordering criterion is serializability [Pap79], which requires that the result of executing a set of transactions be equivalent to (contain the same operations and results as) some execution in which the transactions take place one at a time, and any transactions executed by the same thread take place in program order. Strict serializability imposes the additional requirement that if transaction $A$ completes before $B$ starts in the actual execution, then $A$ must occur before $B$ in the equivalent serial
execution. The intent of this definition is that if external (nontransactional) operations allow one to tell that \( A \) precedes \( B \), then \( A \) must serialize before \( B \). For transactional memory, it seems reasonable to equate external operations with nontransactional memory accesses, and to insist that such accesses occur between the transactions of their respective threads, in program order.

More formally, we define the following strict (asymmetric, irreflexive) ordering relations:

**Program order**, \( <_p \), is a union of disjoint total orders, one per thread. We say \( a <_p b \) iff \( a \) and \( b \) are executed by the same thread, and \( a \) comes before \( b \) in the natural sequential order of the language in which the program is written. Because transactions do not overlap, if transactions \( A \) and \( B \) are executed by the same thread, we necessarily have either \( \forall a \in A, b \in B : a <_p b \) or \( \forall a \in A, b \in B : b <_p a \).

For convenience, we will sometimes say \( A <_p B \) or \( B <_p A \). For \( a \notin B \), we may even say \( a <_p B \) or \( B <_p a \).

**Transaction order**, \( <_t \), is a total order on all transactions, across all threads. It is consistent with program order. That is, \( A <_p B \implies A <_t B \). For convenience, if \( a \in A, b \in B, \) and \( A <_t B \), we will sometimes say \( a <_t b \).

**Strict serial order**, \( <_{ss} \), is a partial order on memory accesses. It is consistent with transaction order. It also orders nontransactional accesses with respect to preceding and following transactions of the same thread. Formally, for all accesses \( a \) and \( c \) in an execution history \( H \), we say \( a <_{ss} c \) iff at least one of the following holds:

1. \( a <_t c \);
2. \( \exists A \in T_H : (a \in A \land A <_p c) \);
3. \( \exists C \in T_H : (a <_p C \land c \in C) \);
4. \( \exists \text{access} b \in H : a <_{ss} b <_{ss} c \).

Note that this definition does *not* relate accesses performed by a given thread between transactions.
An execution with program order \( <_p \) is said to be strictly serializable if there exists a transaction order \( <_t \) that together with \( <_p \) induces a strict serial order \( <_{ss} \) that permits all the values returned by reads in the execution, as defined in the following subsection. A TM implementation is said to be strictly serializable if all of its executions are strictly serializable.

### 7.3.2 Values read

To avoid the need for special cases, we assume that each thread history \( H^i \) begins with an initial transaction \( T^i_0 \), that \( T^i_0 \) writes values to all statically initialized data, and that for all \( i > 0 \), \( T^i_0 <_t T^i_0 \).

We say a memory access \( b \) intervenes between \( a \) and \( c \) iff \( a <_p b \lor a <_{ss} b \) and \( b <_p c \lor b <_{ss} c \). Read \( r \) is then permitted to return the value written by write \( w \) only if \( r \) and \( w \) access the same location \( l \) and either (1) \( r \) and \( w \) are incomparable under both program and strict serial order or (2) \( w <_p r \lor w <_{ss} r \) and there is no intervening write of \( l \) between \( w \) and \( r \).

For the sake of generality across languages and machines, we have kept these rules deliberately parsimonious. In a practical language definition, we would expect them to be augmented with additional rules to capture such concepts as coherence and causality [MPA05]. For example, in

\[
\text{initially } x == 0 \\
T1: x = 1 \quad || \quad T2: \text{atomic}(a = x); \text{atomic}(b = x)
\]

the language memory model will probably insist that if \( a == 1 \) when \( T2 \) completes, then \( b != 0 \); that is, \( x \) is coherent. Likewise, in
initially \( x == y == 0 \)

\[ T_1: x = 1 \quad || \quad T_2: \text{if} (x == 1) \quad y = 1 \quad || \quad T_3: \text{atomic}\{a = y\}; \text{atomic}\{b = x\} \]

the language memory model will probably insist that if \( a == 1 \) when \( T_3 \) completes, then \( b == 1 \) also; that is, the system is causally consistent. Deciding on a complete set of such rules is a Herculean and potentially controversial task (witness the memory models for Java and C++); we do not attempt it here. For any given choice of underlying model, however, we argue that strict serializability, as defined above, is equivalent to SLA. If an execution is strictly serializable, then it is equivalent by definition to some execution in which transactions occur in a serial order (\( <_t \)) consistent with program order and with nontransactional operations. This serial order is trivially equivalent to an execution in which transactions acquire and release a global lock. Conversely, if transactions acquire and release a global lock, they are guaranteed to execute one at a time, in an order consistent with program order. Moreover given a common underlying memory model, SLA and strict serial order will impose identical constraints on the ordering of nontransactional accesses relative to transactions.

### 7.3.3 Selective Strictness

A program that accesses shared data only within transactions can be considered properly synchronized, and can safely run on any TM system that respects \( <_t \). A program \( P \) that sometimes accesses shared data outside transactions, but that is nonetheless data-race-free with respect to \( <_{ss} \) (Abadi et al. refer to this property as violation-freedom [ABHI08]) can also be considered properly synchronized, and can safely run on any TM system \( S \) that respects \( <_{ss} \). Transactions in \( P \) that begin and end, respectively, a region of data-race-free nontransactional use are referred to as privatization
and publication operations, and \( S \) is said to be *publication and privatization safe* with respect to \( <_{ss} \).

Unfortunately, most existing TM implementations are not publication and privatization safe with respect to strict serializability, and modifying them to be so would incur nontrivial costs. It is not yet clear whether these costs will be considered an acceptable price to pay for simple semantics. It therefore seems prudent to consider weaker semantics with cheaper implementations. Menon et al. [MBS+08a] approach this task by defining more complex locking protocols that relax the serialization of transactions. In contrast, we propose a weaker ordering, *selective strict serializability*, that retains the serialization of transactions, but relaxes the ordering of nontransactional accesses with respect to transactions. Specifically, we assume a set of *acquiring* (privatizing) transactions \( A_H \subseteq T_H \) and a set of *releasing* (publishing) transactions \( R_H \subseteq T_H \). \( A_H \) and \( R_H \) are not required to be disjoint, nor are they necessarily proper subsets of \( T_H \). We require that the initial transaction \( T_0^i \) be acquiring for all \( i \); aside from this, it is permissible for all transactions, or none, to be identified as acquiring and/or releasing.

**Selective strict serial order**, \( <_{sss} \), is a partial order on memory accesses. Like strict serial order, it is consistent with transaction order. Unlike strict serial order, it orders nontransactional accesses only with respect to preceding acquiring transactions and subsequent releasing transactions of the same thread (and, transitively, transactions with which those are ordered). Formally, for all accesses \( a, c \in H \), we say \( a <_{sss} c \) iff at least one of the following holds: (1) \( a <_t c \); (2) \( \exists A \in A_H : (a \in A \land A <_p c) \); (3) \( \exists C \in R_H : (a <_p C \land c \in C) \); (4) \( \exists \) access \( b \in H : a <_{sss} b <_{sss} c \).
Listing 7.3: “Publication” by antidependence (adapted from Menon et al. [MBS+08a]). If \( B \) is a releasing transaction, selective strict serializability guarantees that the write of \( x \) is visible to \( D \) (\( i == 1 \), which makes sense). In addition, if \( k == \text{false} \), then \( B \) must have serialized before \( F \), and thus \( j \) must equal 1 as well. Unfortunately, it is difficult for an STM implementation to notice a conflict between \( B \) and \( F \) if \( B \) commits before \( F \) writes \( T3\_used\_x \), and undesirable to outlaw the prefetch of \( x \).

### 7.3.4 Asymmetric Flow

Strict serializability, whether “always on” or selective, shares a problem with the DLA (disjoint lock atomicity) semantics of Menon et al. [MBS+08a]: it requires the useless but expensive guarantee illustrated in Listing 7.3. Specifically, \( a <_p B <_t F \Rightarrow a <_ss F \), even if \( B \) and \( F \) are ordered only by antidependence.

We can permit a cheaper implementation if we define a weaker ordering, selective flow serializability, that requires nontransactional-to-transactional ordering only when transactions are related by a true (flow) dependence:

**Flow order**, \( <_f \subset <_t \), is a partial order on transactions. We say that \( A <_f C \) if there exists a transaction \( B \) and a location \( l \) such that \( A <_t B \), \( A \) writes \( l \), \( B \) reads \( l \), and \( B = C \lor B <_t C \).

**Selective flow serial order**, \( <_{sfs} \subset <_{sss} \), is a partial order on memory accesses. It is consistent with transaction order. It does not order nontransactional accesses
Listing 7.4: Unnecessary ordering in ALA. When $B$ commits and $D$ reads $x_{pub}$, ALA forces $D$ to abort (as it should, since $x_{pub}$ has to be logically acquired as of the beginning of $D$, and that is impossible once $B$ has committed). When $D$ commits and $F$ reads $y$, ALA will likewise force $F$ to abort (since it is flow dependent on a committed transaction with a later start time), though one could, in principle, simply serialize $F$ after $D$. With SFS, the programmer would presumably mark $B$ but not $D$ as a releasing transaction, and $F$ would not have to abort.

with respect to a subsequent releasing transaction $B$, but rather with respect to transactions that have a flow dependence on $B$. Formally, $\forall$ accesses $a, c \in H$, we say $a <_{sfs} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists A \in A : (a \in A \land A <_p c)$; (3) $\exists B \in R, C \in T : (a <_p B <_f C \land c \in C)$; (4) $\exists$ access $b \in H : a <_{sfs} b <_{sfs} c$. It is not difficult to see that $<_{sfs} \subset <_{sss}$.

The sets of values that reads are permitted to return under selective strict and flow serial orders are defined the same as under strict serial order, but with $<_{sss}$ and $<_{sfs}$, respectively, substituted for $<_{ss}$. These induce corresponding definitions of SSS and SFS executions and TM implementations. In comparison to ALA, SFS is not defined in terms of locks, and does not force ordering between nontransactional accesses and unrelated transactions (Listing 7.4).
Like ALA, SFS can lead to apparent temporal loops in racy programs. In Listing 7.3, for example, both semantics allow \( k == \text{false} \) and \( j == 0 \), even if \( B \) is a releasing transaction. Naïvely, this output would seem to suggest that \( a < B < F < a \). ALA breaks the loop by saying that \( B \) and \( F \) are not really ordered. SFS breaks the loop by saying that \( a \) and \( F \) are not really ordered. Which of these is preferable is perhaps a matter of taste.

It should be emphasized that private (nontransactional) use of data that are sometimes accessed by other threads is safe only when data-race-free. To be confident that a program is correct, the programmer must identify the transactions that serve to eliminate races. This may sometimes be a difficult task, but it is the inherent and unavoidable cost of privatization, even under SLA. Once it has been paid, the extra effort required to label acquiring and releasing transactions is essentially trivial.

### 7.4 Implementing SSS and SFS Systems

Both the doomed transaction problem and the undo log variant of the delayed cleanup problem (footnote 2) involve abortive attempts to execute transactions. Since these attempts play no role in the (user-level) semantics of Section 7.3, we need to extend our formalism. For discussion of implementation-level issues, we extend thread histories (as in our previous work [Sco06]) to include \begin{environment}, \commit, and \abort operations. None of these takes an argument. \begin{environment} \abort \return no value. \commit \return a Boolean indication of success. Each thread history is assumed to be of the form \((read | write)^* \begin{environment} \commit \begin{environment} (read | write)^* \end{environment} \begin{environment} (read | write)^* (commit | abort) \end{environment} \begin{environment} (read | write)^* (for simplicity, we assume that nested transactions are subsumed into their parent). A transaction comprises the sequence of operations
from begin through the first subsequent commit or abort in program order. A transaction is said to succeed iff it ends with a commit that returns true.

With this extended definition, for \(<_g \in \{<_ss, <_sss, <_sfs\}\), a read \(r\) is permitted to return the value written by a write \(w\) iff they access the same location \(l\) and (1) \(w\) does not belong to an unsuccessful transaction, and \(r\) and \(w\) are incomparable under both \(<_p\) and \(<_g\); (2) \(w\) does not belong to an unsuccessful transaction, \(w <_p r\) or \(w <_g r\), and there is no intervening write of \(l\) between \(w\) and \(r\); or (3) \(w\) and \(r\) belong to the same transaction, \(w <_p r\), and there is no intervening write of \(l\) between \(w\) and \(r\). A memory access \(b\) intervenes between \(a\) and \(c\) iff \(a <_p b \lor a <_g b\) and \(b <_p c \lor b <_g c\) and (a) \(b\) and \(c\) belong to the same transaction, or (b) neither \(a\) nor \(b\) belongs to an unsuccessful transaction. These rules are roughly equivalent to those of Guerraoui and Kapalka [GK08], but simplified to merge request and reply events and to assume that histories are complete (i.e., that every transaction eventually commits or aborts), and extended to accommodate nontransactional accesses. In particular, we maintain their requirement that transactions appear to occur in serial order (\(<_t\)), and that writes in unsuccessful transactions are never externally visible.

We assume that every correct STM implementation suggests, for every execution, a (partial or total) natural order \(<_n\) on transactions that is consistent with some \(<_t\) that (together with \(<_p\)) can explain the execution’s reads. For the implementation to ensure SSS semantics, it must provide publication and privatization safety only around selected releasing and acquiring transactions, respectively. To ensure SFS semantics, it must provide the same privatization safety, but need only provide publication safety beyond selected flow-ordered transactions.
7.4.1 Preventing the Doomed Transaction Problem

If transactions $D$ and $A$ conflict, and $A$ commits, STM runtimes typically do not immediately interrupt $D$’s execution. Instead, $D$ is responsible for detecting the conflict, rolling itself back, and restarting. This may occur as early as $D$’s next transactional read or write, or as late as $D$’s commit point. Until the conflict is detected, $D$ is said to execute in a “doomed” state. If $A$ privatizes a region accessed by $D$, subsequent writes to that region may race with concurrent transactional accesses by $D$.

The doomed transaction problem occurs when an STM implementation admits an execution containing a failed transaction $D$, a nontransactional write $w$, an acquiring transaction $A <_p w$, and a natural transaction order $<_n$ such that any $<_t$ consistent with $<_n$, when combined with $<_p$, induces a global (SS, SSS, SFS) order $<_g$ that fails to explain a value read in $D$—specifically, when there are dependences that force $D <_t A$, but there exists a read $r \in D$ that returns the value written by $w$, despite the fact that $D <_g A <_g w$.

In managed code, it appears possible to use run-time sandboxing to contain any erroneous behavior in doomed transactions [HPST06; MBS+08a]. For unmanaged code, or as an alternative for managed environments, we present three mechanisms to avoid the inconsistencies that give rise to the problem in the first place.

**Quiescence.** A transactional fence [SMDS07] blocks the caller until all active transactions have committed or aborted and cleaned up. This means that a fence $f$ participates in the natural order $<_n$ on transactions and in program order $<_p$ for its thread. Since $D <_t A <_t f <_p w$, and $f$ waits for $D$ to clean up, we are guaranteed that the implementation respects $D <_g w$. 
Polling. Polling for the presence of privatizers can tell a transaction when it needs to check to see if it is doomed. This mechanism requires every privatizing transaction to atomically increment a global counter (e.g., with a fetch-and-increment [fai] instruction) that is polled by every transaction, on every read of shared data. When a transaction reads a new value of the counter, it validates its read set and, if doomed, aborts before it can see an inconsistency caused by a private write. Pseudocode for this mechanism appears in Listing 7.5.

Like a transactional fence, the increment $c$ of priv_count participates in $<_n$. Suppose $D$ contains a read $r$ that sees (incorrectly) a value written by $w$, with $D <_t A <_t c <_p w$. Since $c$ increments priv_count and $D$ reads priv_count as part of every TxRead, $D$ must abort before completing $r$, a contradiction.

Timestamp Polling. In a timestamp-based STM like TL2 [DSS06], every writer increments a global timestamp. If all transactions are writers (and hence all update the global timestamp), polling this timestamp prevents the doomed transaction problem, using the same argument as above. When a read-only transaction $A$ may privatize (privatization by antidependence), it does so by reading a value written by a previous non-privatizing transaction. Thus it suffices to observe that while $A$ may not increment the global timestamp, $A$ is ordered after some other transaction $W$ ($W <_n A$) that committed a write in order for $A$’s privatization to succeed. If $D$ is doomed because of

```
TxBegin(desc)          TxRead(&addr, &val, desc)
   ...
   desc->priv_cache = priv_count  t = priv_count
   if (t != desc->priv_cache)    validate()
   Acquire()                     fai(&priv_count)
   fai(&priv_count)
   desc->priv_cache = t
```

Listing 7.5: Polling to detect doomed transactions.
the privatization (and still active), it must read a value written by \( W \). \( W \)'s increment of the global timestamp is sufficient to force a polling-based abort in \( D \) prior to any use of an inconsistent value.

### 7.4.2 Preventing the Delayed Cleanup Problem

The delayed cleanup problem occurs when an STM implementation admits an execution containing a successful transaction \( D \) whose cleanup is delayed,\(^3\) a nontransactional read \( r \), an acquiring transaction \( A <_p r \), and a natural transaction order \( <_n \) such that any \( <_t \) consistent with \( <_n \), when combined with \( <_p \), induces a global (SS, SSS, SFS) order \( <_g \) that fails to explain the value read by \( r \)—specifically, when there are dependences that force \( D <_t A \), but \( r \) returns the value from some write \( w \) despite an intervening write \( w' \in D \) to the same location. We propose three mechanisms to avoid this problem.

**Quiescence.** As before, let \( A \) be immediately followed by a transactional fence \( f \), and let \( D \) commit before \( A \). Since \( D <_t A <_t f <_p r \), and \( f \) waits for \( D \) to clean up, we are guaranteed that the implementation respects \( D <_g r \).

**Optimized Commit Linearization.** Menon *et al.* [*MBS*\(^+\)08a] describe a commit linearization mechanism in which all transactions must increment global counters at transaction begin and while committing. Unfortunately, forcing read-only transactions to modify shared data has a serious performance cost. To avoid this cost, we propose an alternative implementation of commit linearization in Listing 7.6. The implementation

\(^{3}\)In redo log-based STM algorithms, cleanup entails replaying speculative writes and releasing ownership of locations. As noted in footnote 2, undo log-based STM algorithms have an analogous problem, which we ignore here.
Listing 7.6: An implementation of commit linearization in which read-only transactions do not update shared metadata.

is inspired by the classic ticket lock: writer transactions increment the global timestamp, clean up, and then increment a second `cleanups` counter in order. Readers read the timestamp and then wait for `cleanups` to catch up.

We argue that this mechanism is privatization safe with respect to (even non-selective) strict serializability. If writer \( D \) precedes writer \( A \) in natural order but has yet to clean up, then \( D \) will not yet have updated the `cleanups` counter, and \( A \)'s `TxCommit` operation will wait for it. Any subsequent read in \( A \)'s thread can be guaranteed that \( D \) has completed.

Suppose that reader \( A \) privatizes by antidependence. If \( D \) increments the global timestamp before \( A \) begins, \( A \) must wait in `TxBegin` for \( D \) to clean up, avoiding the problem. If \( D \) is still active when \( A \) begins, there must exist some other writer \( W \) \((W <_n A)\) that committed a write in order for \( A \)'s privatization to succeed. Clearly \( D \neq W \), or else \( D \)'s write of the location in the antidependence would have forced \( A \) to abort. Moreover, since the program is data-race-free, \( D <_n W \). For \( D \) to still be active when \( A \) begins we must have \( W \) still active when \( A \) begins, a contradiction, since \( W \) writes a location that \( A \) reads, and \( A \) does not abort.
Commit Fence. Our commit fence mechanism combines the best features of the transactional fence and commit linearization. As in the transactional fence, there is no single global variable that is accessed by all committing writer transactions. As in commit linearization, only committing transactions can cause a privatizer to delay. Pseudocode for the mechanism appears in Listing 7.7.

The commit fence ensures that any transaction sets an indicator before acquiring locks, and unsets the indicator after releasing locks. At its acquire fence, a privatizing transaction samples all transactions’ indicators, and waits until it observes every indicator in an unset state. This commit fence $c$ provides privatization safety as above: if $D$ accesses data privatized by $A$, and if $D \prec_t A$, then $D$ must update the commit fence before $A$ completes its commit sequence. Since $A \prec_p c$, and $c$ observes $D$’s in-flight modifications, $c$ will not return until $D$ completes, and thus $D \prec_g c$.

Unlike the full transactional fence, this mechanism does not prevent the doomed transaction problem. Like the transactional fence, it can cause an acquirer $A$ to wait on a committing, nonconflicting transaction $B$ even when $A \prec_t B$. However, as in commit linearization, $A$ will block only for committing transactions, never for in-flight transactions.
### 7.4.3 Preventing Publication Errors

Under SSS, publication safety can be expressed as the condition that if $w <_p R <_t T$, where $R \in \mathcal{R}$, then $w <_{\text{sss}} T$, even if $T$ reads the location written by $w$. We propose two release implementations that guarantee this condition.

**Quiescence.** Placing a transactional fence between a nontransactional access and a subsequent publishing transaction prevents the publication problem. Suppose $w <_p R$, where $w$ is a nontransactional write of location $l$ and $R \in \mathcal{R}$. The publication problem manifests when some transaction $T$ prefetches $l$ before $w$ writes it, but $R <_n T$. If $T$ begins before $w$, a fence between $w$ and $R$ forces $T$ to complete before $R$ begins, so $R \not<_n T$.

**Polling.** Polling may also be used, as shown in Listing 7.8, to prevent the publication problem. Instead of having a publisher wait for all active transactions to complete, each active transaction $T$ checks at each read (and at $\text{TxCommit}$) to see whether a release operation $e$ has occurred since $T$ began execution. If $e <_p R <_n T$ and $T$ is successful, we are guaranteed that $T$ was not active at the time $e$ occurred, and $T$ could not have prefetched any published datum.

### 7.4.4 Preventing Flow Publication Errors

Flow-serializable publication safety requires only that if $w <_p R <_f T$, where $R \in \mathcal{R}$, then $w <_{\text{sfs}} T$. That is, the existence of $R$ need not cause $T$ to abort unless $T$ reads something $R$ wrote. Menon et al. use timestamps to achieve ALA semantics. Their timestamps, however, are not TL2 timestamps, as they are assigned at begin time,
Listing 7.8: Polling to detect publication.

even for read-only transactions. We briefly argue that TL2 timestamps [DSS06] provide SFS (Listing 7.9).

Let us assume (due perhaps to compiler reordering) that $T$ races with $R$ to prefetch $l$. This indicates that $T$ could not start after $R$, and thus $T.\text{start} \leq R.\text{start}$. If it also holds that $R <_{f} T$, then when $R$ acquires timestamp $t$ at commit time, $t > R.\text{start} \geq T.\text{start}$. $R$ subsequently writes $t$ into all lock words that it holds. If $R <_{f} T$, $T$ must read some location written by $R$. During $T$’s call to $\text{TxRead}$ for $l$, the test of the timestamp will cause $T$ to abort, restart, and re-read $l$ after $R$. Alternately, if $T$ reads the lock covering $l$ before $R$ commits, then either $T <_{s} R$, or $T$ will abort at its next validation. In either case, $T$ cannot be ordered both before $w$ and after $R$.

We note that TL2 timestamps reduce the scalability of non-publishing, non-privatizing transactions when compared to the extendable timestamps of Riegel et al. [RFF07]. They also enforce more ordering than the timestamps of Menon et al. [MBS+08a],
Listing 7.9: TL2-style timestamps.

which do not abort a transaction $T$ that reads a value written by a publisher that started
(but did not commit) before $T$ began. Finally, TL2-style timestamps preclude partial
rollback for closed nested transactions: If $B$ reads a value that $C$ writes, and $B$ is nested
within $A$, then the requirement for $B$ to order after $C$ necessitates that $A$ order after $C$
as well. Even if all accesses prior to $B$ in $A$ do not conflict with $C$, $A$ must restart to
acquire a start time compatible with ordering after $C$. 

```c
TxStart(desc)
... 
desc->start = timestamp;
TxCommit(desc)
... // acquire locks
endtime = get_timestamp();
if (validate())
    replay_redo_log()
    foreach (lock in writeset)
        lock.releaseAtVersion(endtime)
... 
TxRead(addr* addr, addr* val, desc)
  // addr not in write set
  oreo o = get_orec(addr);
  if (o.locked || o.timestamp > desc->start)
      abort()
  ... // read value
  if (o != get_orec(addr))
      abort()
  ... // log oreo, return value
```
7.5 Evaluation

In this section, we evaluate the role that selective semantics can play in reducing transaction latency without compromising correctness. We use targeted microbenchmarks to approximate three common idioms for privatization and publication. All experiments were conducted on an 8-core (32-thread), 1.0 GHz Sun T1000 (Niagara) chip multiprocessor running Solaris 10. All benchmarks were written in C++ and compiled with g++ version 4.1.1 using –O3 optimizations. Data points are the average of five trials.

7.5.1 STM Runtime Configuration

We use a word-based STM with 1 million ownership records, commit-time locking, and buffered updates. Our STM uses timestamps to avoid validation overhead, and unless otherwise noted, the timestamps employ a extendable time base scheme [RFF07] to safely allow nonconflicting transactions to ignore some timestamp-based aborts. From this STM, we derive 10 runtimes:

- SLA. Uses the start and commit linearization of Menon et al. [MBS+08a], and polls the global timestamp on reads to avoid the doomed transaction problem.

- SSS-FF. Uses transactional fences before publishing transactions and after privatizing transactions.

- SSS-FL. Uses our commit linearization for privatization, polls the global timestamp to avoid the doomed transaction problem, and uses transactional fences before publishing transactions.
• SSS-PF. Uses polling for publication safety, and transactional fences for privatization.

• SSS-PL. Uses polling for publication safety, commit linearization for privatization, and polling to avoid the doomed transaction problem.

• SSS-PC. Uses polling for publication safety, commit fences for privatization, and polling to avoid the doomed transaction problem.

• SSS-FC. Uses commit fences for privatization, polls the global timestamp to avoid the doomed transaction problem, and uses transactional fences before publishing transactions.

• ALA. Uses TL2-style timestamps for publication safety, and commit linearization with polling of the timestamp for privatization.

• SFS-TL. Identical to ALA, but maintains a privatization counter, separate from the timestamp, to avoid the doomed transaction problem.

• SFS-TF. Uses TL2-style timestamps for publication safety, and transactional fences for privatization.

### 7.5.2 Phased Privatization

In some applications, program structure, such as barriers and thread join points, ensures that all threads agree that a datum is public or private [SSDM07]. Since these phase boundaries are ordered globally, and with respect to $<_t$, no additional instrumentation is required for correctness on acquiring and releasing transactions. However, as in applications with no privatization or publication, ALA or SLA semantics cause ordering latency for all transactions.
We model the transactional phase of a phased workload with a low-contention red-black tree (Figure 7.1(a)). Threads use 20-bit keys and perform 80% lookups, 10% inserts, and 10% removes. We ensure steady state by pre-populating the tree to 50% full.

Since SLA serializes all transactions, it consistently underperforms the other runtimes. Similarly, ALA and other mechanisms that use commit linearization fail to scale as well as mechanisms that do not impose additional ordering on all writers at commit time. However, the SSS-FL, SSS-PL, and SFS-TL curves show that our optimized mechanism for commit linearization, which does not force read-only transactions to increment a global shared counter, achieves better throughput when writing transactions are rare.⁴

Our test platform clearly matters: the Niagara’s single shared L2 provides low-latency write misses for the variables used to provide commit linearization, preventing additional slowdown as the lines holding the commit and cleanup counters bounce between cores. At the same time, the Niagara’s single-issue cores cannot mask overheads due to polling for publication safety. Thus SSS-FF and SFS-TF perform best. Since commit fences require polling to prevent the doomed transaction problem, SSS-FC performs worse than SSS-FF.

### 7.5.3 Worklist Privatization

The privatization problem was first discovered in worklist-based applications, where transactions cooperatively create a task and enqueue it into a nontransactional worklist. When the task is removed from the worklist, the data comprising the task are logically

---

⁴Higher writer ratios show the same separation, with less difference between commit linearization and SLA.
private. Abstractly, these workloads publish by sharing a reference to previously private data, and privatize by removing all shared references to a datum. In the absence of value speculation, these applications admit the privatization problem, but not the publication problem.

To evaluate selective semantics for worklists, we use a producer/consumer benchmark, in which multiple threads cooperatively produce tasks, and then pass the tasks to a consumer thread. We model tasks as red-black trees holding approximately 32 6-bit values, and build tasks using an equal mix of insert, remove, and lookup operations on initially empty trees. Once a tree is found to hold a distinguished value, it is privatized and sent to a nontransactional consumer thread. For the experiment in Figure 7.1(b), the consumer is fast enough that even 32 producers cannot oversaturate it.

Mechanisms that impose excessive ordering (SLA and ALA) or use commit linearization (SSS-FL, SSS-PL, and SFS-TL) perform worst. Furthermore, since transactions are small, and since privatizing a task does not prevent other producers from constructing a new task, the overhead of a transactional fence (SSS-FF and SSS-PF) at privatization time is as low as the commit fence (SSS-PC and SSS-FC). TL2-style timestamps (ALA, SFS-TL, and SFS-TF) decrease scalability. Again, due to the architecture of the Niagara CPU, polling for publication (SSS-PF, SSS-PL, and SSS-PC) or doomed transaction safety (SLA, SSS-FL, SSS-PL, SSS-PC, SSS-FC, ALA, and SFS-TL) increases latency slightly.

### 7.5.4 Indirect Publication

When the value of a shared variable determines whether another location is safe for private access, both the publication and privatization problems can arise. This program-
(a) Transactional phase of a phased privatization workload, modeled as a low contention red-black tree.

(b) Worklist privatization, using transactional producers and a single private consumer.

Figure 7.1: Privatization microbenchmarks.
ming idiom is analogous to locking: the period of private use corresponds to a critical section. We explore it with extensible hashing.

In Figure 7.2, transactions perform 8 puts into a set of 256 hash tables, where each table uses per-bucket sorted linked lists. If transaction $T$ encounters a bucket containing more than 4 elements, it sets a local flag. After committing, $T$ privatizes and then rehashes any flagged hash tables, doubling the bucket count (initially 8). In order to maintain a steady rate of privatization, if a table’s bucket count reaches $2^{13}$, the table is privatized and passed to a worker thread $W$. $W$ replaces the table with an empty 8-bucket table.

Scalability is low, because privatization for rehashing essentially locks the hash tables. Even with 256 tables, the duration of rehashing is significantly longer than the duration of several 8-put transactions, and thus even at two threads, when one thread privatizes a table for rehashing, the other thread is likely to block while attempting to access that table.
The different mechanisms vary only when there is preemption (at 32 worker threads, since there is an additional thread that performs table destruction). At this point, all privatization mechanisms risk waiting for a preempted transaction to resume. The effect is worst for the transactional fence (SSS-FF, SSS-PF, SFS-TF), since it must wait for all active transactions. Our commit linearization (SSS-FL, SSS-PL, ALA, SFS-TL) fares much better, since threads only wait for previously committed writers, who necessarily are at the very end of their execution. SLA linearization lies somewhere in between. Unlike transactional fences (which also avoid the doomed transaction problem), its use of a global timestamp avoids waiting for logically “later” transactions that are still in progress, but unlike commit linearization, it also must wait on “earlier” transactions that have not reached their commit point. The commit fence (SSS-PC, SSS-FC) performs slightly worse than SLA, indicating that waiting on “later” transactions is more expensive than waiting for “earlier” transactions.

The low latency of fence-based publication appears to be an artifact of the indirect publication idiom. At a fence, the releasing transaction waits for all concurrent transactions to commit or abort and clean up. Since some hash tables are effectively locked, most concurrent threads will execute “restart” transactions to spin-wait for the tables to become public. In our lazy STM, such transactions do not hold locks and can rapidly restart, preventing delays at the publication fence.

7.6 Summary

In this chapter we presented an ordering-based semantics for TM. In our specification, traditional strict serializability (SS) takes the place of single lock atomicity (SLA). To reduce the cost of publication and privatization, we proposed selective strict serializability (SSS), which enforces a global order between transactional and nontransactional
accesses of a given thread only when transactions are marked as acquiring or releasing. We also proposed a weaker selective flow serializability (SFS), that enforces release ordering only with respect to transactions that read a location written by the releasing transaction. We described several possible implementations of both SSS and SFS, with informal correctness arguments.

Our preliminary experiments show that by imposing the cost of publication and privatization only when they actually occur, selective ordering of nontransactional accesses can offer significant performance advantages. Given selectivity, there seems to be no compelling argument to relax the serial ordering of transactions (through SFS or ALA). Moreover we suspect that requiring annotations will ultimately help the programmer and compiler to generate race-free code.

Our definition and implementations of SSS make a compelling case for ordering-based semantics instead of lock-based semantics. SSS is simpler to explain to novice programmers than semantics based on prescient lock acquisition; it allows true closed nesting, unlike ALA and SFS; it offers superior performance to single-lock semantics; and it is orthogonal to the underlying STM implementation.
8 Conclusions and Future Directions

The primary motivation for parallel programming is performance. For applications that manipulate large data sets, only parallel computational resources are capable of generating timely results. Across a wide spectrum of disciplines, such as the physical sciences, engineering, medical applications, and social sciences, the greatest problems of our time are being expressed as massive computations. Harnessing parallelism to write efficient programs that generate results rapidly can have a significant impact on our daily lives.

Transactional Memory (TM) promises to help the writers of these applications to develop parallel solutions, by removing the need for programmers to reason about fine-grained synchronization explicitly. By specifying what requires atomicity rather than how to achieve atomicity, TM should simplify the task of harnessing parallelism, so long as it delivers acceptable performance.
8.1 Contributions

In this dissertation, we focused on improving software TM performance. In particular, we (I) presented new approaches to decrease the latency of individual transactions, (II) developed mechanisms to ensure progress and throughput, (III) explored techniques that allow transactions to perform I/O, and (IV) proposed an ordering-based semantics that preserves the serializability of transactions while allowing programs to access data from transactional and nontransactional contexts.

8.1.1 Latency

Scalable STM algorithms appear to require instrumentation on every transactional load and store. In addition, committing a transaction is typically expensive. In STM algorithms that use per-location ownership records, the commit phase incurs overhead related to both the number of locations read and the number of locations written. Additionally, periodic checks by an in-flight transaction, to ensure that it is still able to commit, are also costly due both to their frequency and the cost each individual check (typically linear in the number of locations read).

This dissertation demonstrates that all three of these sources of latency can be reduced. In Chapter 2, we presented a global commit counter heuristic. Unlike other techniques to ensure consistency, our mechanism satisfies the dual goals of ensuring strong safety guarantees for in-flight transactions (that is, supporting opacity [GK08]) while also incurring minimal latency in the common case (only a small number of instructions per read by an in-flight transaction). The effect was an order of magnitude single-thread improvement for some workloads, with most microbenchmarks experiencing more than $2\times$ speedup at all thread counts.
In Chapter 3, we showed that per-access read instrumentation could be further reduced in an STM algorithm that did not use ownership records at all. Our RingSTM algorithm avoids the need for write overhead (metadata acquisition and release) linear in the number of locations written; it also uses fixed-size Bloom filters [Blo70] to reduce the cost of each individual validation operation to a constant, instead of being linear in the number of reads. The result is a $5 \rightarrow 10\%$ speedup (relative to orec-based STM) at low thread counts. At higher thread counts, RingSTM performs on par with orec-based algorithms, while providing stronger progress guarantees and implicit privatization safety.

Furthermore, we showed in Chapter 4 that an optimizing compiler could reduce both the number of validations and the frequency of memory fences for STM algorithms implemented on processors with relaxed memory consistency. Our optimizations can improve orec-based STM runtimes, as well as runtimes like RingSTM and JudoSTM that do not use orecls. Using our techniques, memory fence counts dropped by up to 89%, and latency decreased by up to 20% for some benchmarks. Taken together, our commit counter, RingSTM algorithm, and compiler optimizations greatly reduce latency for transactions, and provide a foundation for a number of promising future directions.

8.1.2 Throughput

Our mechanisms for reducing latency also had a positive impact on throughput. In addition to decreasing latency, our commit counter improved throughput by approximating mixed invalidation [Sco06], a useful but previously unexplored approach to resolving conflicts. In RingSTM, the combination of buffered update, single-RMW
commit, and a global order for all writer transactions resulted in a livelock-free algorithm.

For orec-based STM algorithms, we showed in Chapter 5 that careful engineering could deliver practical livelock-freedom without sacrificing latency. We then presented an approach to starvation avoidance. Our solution, which extends a user-defined priority mechanism, allows transactions to request priority after multiple consecutive aborts. Priority transactions incur latency to make their reads visible to concurrent conflicting writers, through mechanisms that are largely orthogonal to the STM implementation. Unlike previous approaches, our technique requires read visibility only for starving transactions (an improvement over the Greedy policy [GHP05b] and DracoSTM [GC08]), allows nonconflicting writer transactions to commit (unlike RingSTM and JudoSTM), and remains compatible with retry-based condition synchronization (an option unavailable to inevitability-based starvation prevention as proposed by Welc et al. [WSAT08] and Olszewski et al. [OCS07]).

8.1.3 Generality

There is concern that a fast, scalable STM implementation may still not suffice to simplify the task of writing correct parallel programs. As a replacement for critical sections, transactions appear insufficient if they cannot support I/O and other irreversible operations. Similarly, if the semantics of atomic regions are not clearly defined, then programs may ultimately observe races and incorrect termination when data transitions between shared and thread-private states.

In Chapter 6, we presented a number of mechanisms that support irreversible operations through inevitable transactions (that is, transactions that never abort). Our mechanisms enable transactions to execute and commit in parallel with an inevitable
transaction without requiring custom hardware. We also offer a variety of alternatives, each suitable for different workload characteristics (such as duration of inevitable transaction, predictability of reads and writes performed by black-box library code, and frequency of inevitable transactions).

Lastly, in Chapter 7, we presented a natural semantics for transactions, based on serializability. Unlike previous proposals [MBS+08a], our semantics do not require programmers to think in terms of locks. We do, however, require programmers to annotate those transactions that transition data between private and shared states. We show that these annotations allow a variety of efficient implementations, offering as much as a 2× speedup relative to runtimes that provide lock-based semantics. Which implementation has the lowest latency depends on the frequency with which annotated transactions execute.

8.2 Future Directions

Throughout this dissertation, we have assumed that transactions will be used explicitly by a programmer developing general-purpose software, with the transactional library running on existing hardware. We have also assumed an unmanaged runtime without just-in-time compilation, in which the STM algorithm is set at compile time to maximize inlining opportunities. Unfortunately, our results have shown significant dependence on workload characteristics. In this section we briefly consider on-going and future research that questions these assumptions.
8.2.1 Thread-Level Speculation with STM

In the early 2000s, renewed interest in hardware TM grew, in part, out of research into thread-level speculation (TLS) [CMT00; HWC+04; Mar02; MT02; Raj02; RG01; RG02]. As hardware and software TM implementations have grown more powerful, a new question arises: can TM be used to implement efficient thread-level speculation? A particularly appealing option is for a software library to provide TLS for existing multicore systems. A TLS implementation specialized for 2 – 4 cores could also accelerate the execution of sequential code within the nodes of a massively parallel supercomputer.

Unfortunately, oreca-based software TM does not appear to offer sufficiently low latency to make TLS profitable. In particular, the requirement for validation overhead linear in the number of reads, and oreca acquire operations linear in the number of writes, risks making the commit phase of a speculative loop iteration take as long as the speculative iteration itself.

The RingSTM algorithm (and to a lesser extent, the JudoSTM algorithm) promise to change this situation. In ongoing work separate from this dissertation, we have (1) identified differences between sequential program semantics and transactional semantics, which will necessitate sandboxing even when TLS is implemented using an opaque STM algorithm and (2) identified compiler optimizations (some of which build upon those presented in Chapter 4) that optimize fast-path loop iterations. Preliminary results suggest that speedup may be possible for specialized runtimes with 2 – 4 hardware threads.
8.2.2 Custom STM Runtimes for Systems Software

The STM algorithms discussed in this dissertation are general-purpose algorithms, intended to support large transactions and concurrent writes by several threads. A number of critical sections do not require these properties. For example, the Linux kernel uses read-copy-update [McK04] and sequence locks [Lam05] to protect critical sections that are typically read-only. Additionally, mutex locks are often used in place of reader/writer locks when critical sections are small, since the increased cache contention caused by multiple RMW operations per critical section impede scalability even when many critical sections are read-only.

Our recent Transactional Mutex Lock (TML) proposal [SSDS09], which is based on a minimalistic STM using only a single ownership record, appears ideally suited for many workloads in systems software. To date, we have developed a candidate runtime and identified several novel compiler optimizations for TML. While performance is heavily dependent on the underlying architecture, we have found that in some cases (such as on the x86), TML can outperform mutex locks at all thread levels, while offering scalability for workloads with a large percentage of read-only critical sections.

TML’s semantics are slightly weaker than those provided by a mutex (though at least as strong as ALA semantics [MBS+08a]), and TML does require compiler support to instrument individual loads and stores within a lexically-scoped critical section. In ongoing work, we are exploring other cases where TML can be used, to (1) afford read/read concurrency in read-dominated critical sections that occasionally perform writes, (2) replace mutexes for read-dominated workloads that cannot statically determine which critical sections will perform writes, and (3) to increase throughput for write-dominated critical sections of varying lengths.
8.2.3 Dynamic Selection of STM Runtimes

Transactional memory is likely to be added to future versions of the C# and Java programming languages. Similarly, the specifications for the Chapel [CCZ07], Fortress [Sun09], and X10 [CDE05] languages (all developed as part of the DARPA High Productivity Computing Systems program) include some notion of atomic regions. In all cases, these languages are likely to include an adaptive runtime system that dynamically recompiles and optimizes applications.

Throughout this dissertation, we have discussed the need for dynamic adaptivity in STM: RingSTM would benefit from adapting ring and filter sizes, as well as its hash function; orec-based STM would benefit from adapting between eager and lazy acquire (as in ASTM [MSS05]), and again would benefit from dynamic selection of a hash function; the best inevitability mechanism or implementation of acquire and release fences is clearly dependent on workload characteristics. Additionally, we hope to create profiling tools to assist the programmer in statically selecting the best parameters for an STM implementation.

In addition, dynamic selection of a synchronization mechanism to protect atomic regions (to include locks, TML, or any of the STM algorithms described or discussed in this dissertation) would maximize performance for workloads whose characteristics are best suited to a particular algorithm. Some form of just-in-time compilation, dynamic recompilation, or binary rewriting appears necessary to achieve maximum performance in this setting—otherwise, each access will incur multiple conditional branches to determine the appropriate manner of instrumentation [MSH06a], and each accesses will typically require a function call [NWAT08].
8.2.4 Lightweight Hardware Support for Transactional Memory

Research into hardware support for transactional memory (HASTM) [MTC+07; SATJ06; SDS08; SMD+06a; SMD+06b; SSH+07] identified a number of useful mechanisms to decrease STM latency. In many cases, these proposals included modifications to the cache controller, either to buffer speculative writes or to ensure a single-writer protocol.

In work performed separate from this dissertation, we explored an extremely lightweight hardware acceleration mechanism, Alert-on-Update (AOU) [SSD+07]. This mechanism is completely local to a processor core, and serves only to notify the processor of the eviction of a specially-marked cache line. A processor that supports abundant AOU marks can dramatically improve STM throughput by eliminating validation. In a more realistic setting, support for only a single AOU word per hardware context suffices to make some STM implementations nonblocking. AOU can also decrease the cost of polling, thereby lowering the cost of privatization and publication safety. We hope to explore the use of AOU to make RingSTM nonblocking, and thus suitable for use in operating system critical sections, such as interrupt handlers (which cannot tolerate blocking [RHP+07; RRP+07]).

Additional, simple hardware features may improve STM even further. For example, support for small regions of memory with an update protocol may improve RingSTM performance by decreasing the cost of cache misses when accessing the Ring. Hardware maintenance of signatures could decrease the cost of read and write filter management for RingSTM; if those filter updates were ordered with respect to the corresponding memory accesses, such a feature could also accelerate filter-based implementations of priority (Chapter 5) and inevitability (Chapter 6).
Throughout this dissertation, we considered algorithms to reduce STM latency, increase STM throughput, and facilitate the use of TM in programs that perform I/O and nontransactional access to shared memory. Efforts to deliver TM to mainstream programmers are underway, though several challenges remain. Only when those efforts succeed will TM become an effective tool to harness parallelism.
Bibliography


[FFM+07] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying Applications using an


the 2nd ACM SIGPLAN Workshop on Transactional Computing, Portland, OR, August 2007.


[ZR07] Craig Zilles and Ravi Rajwar. Transactional memory and the birthday paradox (brief announcement). In *Proceedings of the 19th ACM Sym-