Efficient Runtime Support for Reliable and Scalable Parallelism

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Minjia Zhang, B.S., M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2016

Dissertation Committee:

Michael D. Bond, Advisor

Atanas Rountev

Radu Teodorescu

Abstract

Commodity hardware is becoming more parallel. To benefit from parallel hardware, software needs to become more parallel as well. Parallel execution of a concurrent program can increase program throughput and reduce program execution time. However, writing concurrent programs that are both correct and scalable is known as notoriously more difficult than writing sequential programs because of the interaction of multi-threaded execution model, synchronization, and the memory hierarchy. Concurrent programming therefore has been traditionally the exclusive domain of a small portion of expert programmers.

To make concurrent programming more accessible, researchers and practitioners have built various *runtime support* to facilitate the programmability, debuggability, performance, and scalability of concurrent programs on multiprocessor machines. This dissertation makes several contributions by introducing novel approaches that provide efficient runtime support on modern commodity multiprocessor hardware to support reliable and scalable concurrent programming with available parallelism.

To address a common challenge in building many types of efficient runtime support—how to efficiently track memory access dependences, the dissertation introduces a novel design that tracks memory dependences in a *relaxed* way which overlaps coordination with program execution to hide coordination latency. The dissertation introduces client analyses to demonstrate the practicality and caveats of the relaxed tracking approach.

Transactional memory is a promising high–level abstraction for shared-memory concurrency. However, it faces challenges in both performance and semantics. The dissertation introduces LarkTM, a low-overhead, scalable *software transactional memory* (STM) system with strong semantics and strong progress guarantees. The dissertation also introduces LarkTM-S, which extends LarkTM by hybridizing optimistic and pessimistic concurrency control, and a variant of LarkTM which is based on relaxed dependence tracking.

Finally, the memory model is at the heart of concurrency semantics. To increase the practicality of strong memory consistency models, the dissertation is the first that identifies the *availability* issue in strong memory models with fail-stop semantics. It introduces a new memory model based on snapshot isolation which has strong, well-defined semantics and devises a memory model enforcement approach that enforces the new memory model while at the same time does best-effort tolerance of consistency exceptions.

To my family.

Acknowledgments

This work could not have been possible without the guidance and inspiration I received from my advisor Michael Bond. Working with Mike was one of the best decisions I have made in my life. Not only he taught me the knowledge necessary for conducting research projects, he has showed me the key qualities one needs to have to become an independent researcher. I am very grateful to Mike for spending hours with me every week to answer all of my questions, going through problems on the board, and even helping me improve my code and writing. Mike has always motivated me to focus on the long run and has given me very useful career advice.

I also thank the other members of my dissertation committee: Professor Nasko Rountev, from whom I learned a lot about compiler, static analysis, and dynamic analysis, and Professor Radu Teodorescu, whose helpful suggestions increased the readability and reduced ambiguity of this dissertation. I thank Professor Dhabaleswar Panda who helped me attend OSU and guided me to work on high-performance distributed computing. I thank our department chair Xiaodong Zhang for helping me see the "big picture" many times when I was facing difficult decisions.

I want to express my gratitude to my fellow OSU students Jipeng Huang, Swarnendu Biswas, Aritra Sengupta, Man Cao, Jake Roemer and Rui Zhang for valuable discussions and research collaborations. I am thankful to all of the friends I met at OSU for making my Ph.D. journey more enjoyable. Thanks to Harry Xu, Bing Ren, Xiangyong Ouyang,

Miao Luo, Hao Wang, Kazuya Sakai, Tian Luo, Mark Arnold, Tony Yan, Jithin Jose, Yang Zhang, Ye Wang, Qinpeng Niu, Di Cao, Fengtao Fan, Linchuan Chen, Yu Su, Yi Wang, Wasiur Radhman, Nusrat Islam, Victor Yang, Martin Kong, Yuan Yuan, Diego Zaccai, Dustin Hoffman, Meisam Fathi Salmi, Xin Tong, Xiaofeng Wu, Ayan Biswas, Zhi Wang, Siyuan Ma, and many others.

I am thankful for guidance and help provided by Kathryn McKinley, Yuxiong He, Sameh Elnikety, and Srikumar Rangarajan, who have supported and mentored me. Microsoft funded my research with three internships.

Finally, I am grateful to my parents Sancai Zhang and Ping Wu for their unconditional love and support. Without them, I would not have come this far.

Vita

August 2016	Ph.D. in Computer Science and Engineering, The Ohio State University University
September 2015	M.S. in Computer Science and Engineering, The Ohio State University University
August 2010	M.S. Computer Science and Technology, Huazhong University of Science and Technology, China
June 2008	B.E. Computer Science and Technology, Huazhong University of Science and Technology, China

Publications

Research Publications

Minjia Zhang, Swarnendu Biswas, Michael D. Bond. Relaxed Dependence Tracking for Parallel Runtime Support. In the 25th International Conference on *Compiler Construction (CC 2016)*, March 2016.

Man Cao, **Minjia Zhang**, Aritra Sengupta, and Michael Bond. Drinking from Both Glasses: Combining Pessimistic and Optimistic Tracking of Cross-Thread Dependences. In the 21st ACM SIGPLAN Symposium on *Principles and Practice of Parallel Programming (PPoPP 2016)*, March 2016.

Swarnendu Biswas, **Minjia Zhang**, Michael D. Bond, and Brandon Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In the ACM SIGPLAN conference on *Systems, Programming, Languages and Applications (OOPSLA 2015)*, October 2015.

Minjia Zhang. SIRe: An Efficient Snapshot Isolation-based Memory Model for Detecting and Tolerating Region Conflicts. In the *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2015)*, October 2015.

Minjia Zhang, Jipeng Huang, Man Cao, and Michael D. Bond. Low-Overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In the 20st ACM SIGPLAN Symposium on *Principles and Practice of Parallel Programming (PPoPP 2015)*, Feburay 2015.

Aritra Sengupta, Swarnendu Biswas, **Minjia Zhang**, Michael D. Bond, and Milind Kulkarni. Hybrid Static-Dynamic Analysis for Statically Bounded Region Serializability. In the the 20th International Conference on *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)* March 2015.

Man Cao, **Minjia Zhang**, and Michael D. Bond. Drinking from Both Glasses: Adaptively Combining Pessimistic and Optimistic Synchronization for Efficient Parallel Runtime Support. In *International Workshop on Dynamic Analysis (WoDet'14)*, March 2014.

Michael D. Bond, Milind Kulkarni, Man Cao, **Minjia Zhang**, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In the ACM SIGPLAN conference on *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2013)*, pages 693-712, March 2013.

Jithin Jose, Hari Subramoni, Miao Luo, **Minjia Zhang**, Jian Huang, Md. Wasi-ur-Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In the 44th *International Conference on Parallel Processing (ICPP '11)*, pages 743-752, September 2011.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Programming Languages and Software Systems	Prof. Michael D. Bond
High-End Systems	Prof. Feng Qin
Distributed Systems Programming	Prof. Gagan Agrawal

Table of Contents

Page

List of Tables

List of Figures

Figure **Page**

Chapter 1: Introduction

Multicore processors have significant computation potential over traditional processors with single core. However, multicore architectures are more complex and have different programming models, development environments, and memory systems. This has driven the need for writing programs that can run in parallel and creating tools that allow developers to easily and efficiently exploit the entire computation power of these parallel hardware. Concurrent programming is a form of programming where multiple computations are executed in overlapping time periods, also called concurrently, instead of sequentially. Concurrent programs can execute in parallel for performance on multicore hardware. However, it puts challenges to programmers to write correct and scalable concurrent programs and has often required expertise.

This dissertation's goal is to make concurrent programming more accessible to programmers through efficient runtime support. Section 1.1 provides introduction and background on problems and challenges in concurrent programming. Section 1.2 presents the thesis statement. Section 1.3 introduces briefly the ways we propose to address the challenges. Section 1.4 describes the contributions and organization of this dissertation. Section 1.5 introduces the meaning and impact of this dissertation.

1.1 Concurrency in Shared-Memory Systems

Software must become more parallel in order to scale to hardware that provides more—instead of faster—cores. In the past, to get a burst of performance improvement, one simply needs to wait for the next generation of processor that has a faster clock speed than the previous generation. In the legendary article, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software", Herb Sutter explains why free and regular performance gains are no longer available, and future software is expected to be written more in a concurrent fashion in order to gain performance benefits [145].

Concurrency and multicore can put huge processing power at our disposal, which makes it possible to create software with more responsiveness, less latency, and increased throughput. However, concurrent programming on multicore is challenging because multicore hardware has a shared-memory architecture, where all cores can make changes to the shared memory, causing *data races*. A data race is generated when a pair of threads in a shared-memory multi-threaded program access the same memory location without any ordering enforced between the two accesses and at least one of them is a write. Data races can cause correctness issues. Though some data races could be benign, many of them can be harmful [22]. In extreme cases, data races can be disastrous [91, 146, 117]. To address these correctness challenges, practioners and researchers have developed better tools to help diagnose or eliminate data race related bugs, to help express parallelism better to that programmers can reason programmers at higher level, etc. These tools face challenges in terms of performance, scalability, and correctness:

- To rigorously analyze and debug a program with data races, researchers and practitioners have built various runtime support systems, such as data race detector, which

detects and reports data races; atomicity violation detector; *record & replay*, which deterministically replay a program's execution. Many of these systems require the ordering information (or dependences) of cross-thread memory accesses (i.e., read–write, write–read, and write–write dependences, which are unknown without explicitly tracking them). However, tracking cross-thread dependences is expensive, and runtime support usually ends up adding an order of magnitude of overhead to the original programs, which limits its usage.

- Programmers have traditionally used locks to restrict the ordering of memory accesses in order to eliminate data races, but it is extremely difficult to use locks to write both fast and correct concurrent programs. An alternative is to use a different programming model: *transactional memory* (TM), which simplifies the conceptual understanding of concurrent programs by treating multiple memory accesses as a transaction and executing them atomically. Though an appealing idea, software transactional memory suffers from significant performance penalties and weak semantics in practice.

- Data races become most problematic when defining the semantics of underlying *memory models* for multi-threaded programs. Existing memory models for concurrent programming languages are still quite incomplete. Mainstream languages such as Java and C++ promise sequential consistency for data-race-free programs, but when the program has data races, it becomes fuzzier on how to support reasonable performance and semantics.

Overall, existing runtime support often adds high overhead or target at weak semantics. Though one might try to solve these issues in programming languages, compilers, or even

hardware, this dissertation tries to address these issues by using novel approaches to build effective runtime support. Various kinds of runtime support have already been built to check or enforce concurrency correctness properties such as atomicity, data-race freedom, and determinism. *Multi-threaded record & replay* enables offline debugging and online replication for concurrent programs [40, 65, 66, 158, 136, 148, 89]. *Software transactional memory* systems built in runtime enforce atomicity of code regions, simplifying and improving programmability of concurrent programming [73, 157, 124, 135, 76, 51, 80, 84, 104, 105, 31, 164]. *Memory model enforcement* provides reasonable programming semantics toward programming ease at the cost of sacrificing performance [17].

Runtime support can affect many issues we have introduced, and they face similar challenge such as how to track memory dependences and maintain metadata efficiently, how to provide atomicity or isolation at the instrumentation and/or code level, and what static and dynamic analyses would help reduce instruction overhead and improve scalability.

1.1.1 Memory Dependence Tracking

Runtime support generally needs to *capture* (i.e., detect or control) dependences of cross-thread memory accesses soundly, which entails using *synchronized* instrumentation to ensure that each program access and its instrumentation execute together atomically (Section 2.1). Furthermore, the instrumentation at potentially racy accesses, which are conflicting accesses to the same variable that are not ordered by synchronization operations, must perform synchronization in order to ensure that thread interleavings are properly captured. This synchronized instrumentation slows execution substantially, often by about an order of magnitude [58, 63, 64, 149].

1.1.2 Concurrent Programming Models

Even with efficient runtime support, perhaps the biggest challenge of concurrent programming is that it is hard to get it right and make it fast at the same time. This is because the programming model, meaning the model in programmers' mind that they use to reason about their programs, is quite different from sequential programs. Furthermore, data races make it difficult to reason about what it means to be correct. Below are some of the programming models for dealing with data races, starting from the most traditional one.

Locking. Lock-based synchronization has been used traditionally to coordinate access to shared-memory data. It works through the principle of mutual exclusion. A thread that owns a lock can be sure that it is the only one to access the associated shared memory data. Typically, a programmer will use one lock per data structure to keep the synchronization simple. Unfortunately, such a coarse-grained locking can lead to serialization on high-contention data structures and therefore severely affect scalability. Another solution is to use fine-grained locking where more than one lock is associated with a data structure. However, find-grained locking can also easily increase the complexity for operations that access shared data, which can easily lead to problems such as deadlock, atomicity violation, priority inversion, and high locking overhead.

Lock-free Programming. An alternative form to lock-based programming is *lock-free programming*. Typically, it involves using atomic primitives provided by hardware such as *compare-and-swap* (CAS) instruction directly to perform synchronization instead of using them to build locking primitives and use those primitives to write concurrent programs.

While lock-free programming can avoid issues with locks such as deadlocks and priority inversion, it still can cause other problems such as livelock and starvation. And although lock-free programming can lead to programs that are more scalable than lock-based programs, in practice concurrent lock-free programming is known as to be much harder for programmers to understand and reason about than lock-based programming. Also, it is difficult to compose two different operations into one atomic action with lock-free programming.

Transactional Memory. Another alternative is *transactional memory* (TM) [79, 73]. In the TM model, programs specify *atomic* regions of code, which the TM system executes speculatively as transactions and ensures serializability of transactions. To ensure serializability, the system detects conflicting transactions, rolls back their state, and re-executes them if necessary.

TM is not a panacea. TM does not help if atomicity is specified incorrectly or too conservatively; it does not help with specifying ordering constraints; and it does not handle irrevocable operations such as I/O well. However, TM has significant potential to improve productivity, reliability, and scalability by allowing programmers to specify atomicity with the ease of coarse-grained locks while providing the scalability of fine-grained locks [115]. TM also enables systems features such as support for speculative optimizations [110].

Despite these appealing benefits, TM is not widely used because it is *impractical*. Existing software transactional memory (STM) are impractical because they provide weak semantics and poor performance—or else strong semantics and even worse performance—leading some researchers to question the viability of STM and call it a "research toy" [55, 37, 160]. STMs are slow largely because *concurrency control* (detecting and resolving

conflicts) is expensive. Concurrency control adds synchronization (i.e., atomic instructions and/or memory fences) even when transactions do not conflict. Furthermore, STMs that provide *strong atomicity* (atomicity of transactions with respect to non-transactional accesses) slow the entire program, not just transactions. *Hardware* TM (HTM) is efficient, but manufacturers have been reluctant to make dramatic changes to already-complex cache coherence protocols, caches, and memories. Recent HTM support is limited, still relying on efficient *software* TM (STM) support (Section 7.2).

1.1.3 Memory Consistency Models

Giving the asynchronous behavior of threads and the hierarchical structure of the memory system, one challenging problem in concurrent programming is: what should be the value retrieved by a read of a shared memory location, i.e., is it the correct and expected data value? In a sequential program, it can be taken for granted that a read operation on a memory location will read the last value previously written at the same location. In sequential codes the concept of last write operation is precisely defined by program order, namely, the order in which the read/write operations appear in the program. In a multicore shared-memory system, read and write operations executed by different threads are not naturally related by program order. What makes things even more complex is that in order to achieve high performance, compilers and architectures for shared-memory programs perform optimizations, such as eliminating and reordering memory accesses, assuming no dependences with concurrent threads.

Though these optimizations may introduce unintended consequences, they are restricted from reordering across synchronization operations (e.g., lock acquire and release), and they allow programming languages and hardware to provide strong guarantees—sequential

consistency (SC), as well as serializability of code regions bounded by synchronization operations—for program executions that are *free of data races*. However, for executions with data races, compiler and hardware optimizations lead to erroneous, unpredictable, often undefined behavior [4, 21]. These guarantees are formalized in the *DRF0* memory consistency model [6], which ensures strong semantics for data-race-free executions only. Java and C++ and other programming languages provide variants of DRF0 [23, 100]. As a result, the state-of-the-art of memory models is still quite incomplete.

1.2 Thesis Statement

This dissertation seeks to address the aforementioned challenges arising in tracking memory dependences, concurrent programming models, and memory models in shared-memory systems, as outlined in Sections 1.1.1, 1.1.2, and 1.1.3 via designing and building efficient, scalable, and realiable runtime support that uses novel concurrency control techniques and mechanisms. This dissertation provides evidence to support the following statement:

Thesis statement: *Using novel concurrency control techniques and mechanisms can make runtime support for memory dependence tracking, programming models, and memory models substantially more efficient, powerful, and scalable.*

1.3 Our Approach

The trade-offs between performance, scalability, and correctness have created challenges for building effective runtime for parallelism. To address these challenges, we introduce the following approaches:

Relaxed dependence tracking. Tracking cross-thread dependences is the key component for building many runtime support. However, it is also very expensive due to synchronization costs. The dissertation explores the potential for reducing the runtime overhead of tracking dependences by *relaxing the requirement that runtime support must track all dependences accurately*—while preserving the runtime support's guarantees and adhering to the language's semantics. We introduce *relaxed dependence tracking* (RT), which enables a thread to continue executing past a memory access involved in a cross-thread dependence, without accurately tracking the dependence. Our design of RT targets dependence tracking based on so-called *biased reader–writer locking* [123, 82, 126, 149, 80, 129, 32, 28] (Section 2.1.1), which avoids the costs of reacquiring a lock for non-conflicting accesses, but incurs latency at conflicting accesses in order to perform *coordination* among conflicting threads (Section 2.1.1.1). The high cost of coordination provides both a challenge and an opportunity for RT to hide this latency, by relaxing the tracking of dependences at accesses involved in dependences. Apart from designing RT, we also present two kinds of runtime support that use RT in this dissertation: (1) an *RT-based dependence recorder* (Section 3.2) and (2) an *RT-based software transactional memory (STM) system* (Section 5.1). Overall, we demonstrate the potential for using novel mechanisms to address a key performance bottleneck of runtime support for concurrent programs.

Programming ease with STM. TM gives the programmers the illusion of serial execution of transactions, and thus helps programmers reason about transaction execution serially because no other transaction will perform any conflicting operation. To improve the pracaticality of TM, we introduce a novel STM called *LarkTM* that provides low instrumentation costs. At the same time, its design naturally guarantees progress and strong semantics.

Three key features distinguish LarkTM from existing STMs. First, it uses *biased* per-object, reader–writer locks (Section 2.1), which a thread relinquishes only when needed by another thread performing a conflicting access—making non-conflicting accesses fast but requiring threads to *coordinate* when accesses conflict. Second, LarkTM detects and resolves transactional conflicts (conflicts between transactions or between a transaction and non-transactional access) when threads coordinate, enabling flexible conflict resolution that guarantees progress. Third, LarkTM provides *strong atomicity* semantics with low overhead by acquiring its low-overhead locks at both transactional and non-transactional accesses.

This basic approach, which we call *LarkTM-O*, adds low single-thread overhead and scales well under low contention. But scalability suffers under higher contention due to the high cost of threads coordinating. We design an *adaptive* version of LarkTM called *LarkTM-S* that handles high-contention accesses, identified by profiling, using different concurrency control mechanisms. We show that both LarkTM-O and LarkTM-S outperform existing high-performance STMs, and our work takes a concrete step toward making concurrent programming easier.

Memory models with fail-stop semantics. A memory model specifies how memory behaves with respect to shared memory read and write operations. From the programmer's point of view, a memory model enables correct reasoning about the memory operations in a program. From the system designer's point of view, a memory model specifies acceptable memory behaviors for the system. As such, the memory model affects many aspects of concurrent programming, including the design of programming languages, compilers, and the underlying hardware. Existing mainstream memory models have weak semantics, and

researchers have proposed memory consistency models that provide strong semantics for all program executions (i.e., racy and non-racy executions) (more details in Section 2.3).

Existing strong memory models have limitations: they either incur high overhead or require custom hardware, or they add relatively low overhead but suffers from high complexity or poor *availability* (i.e., the ability to execute a program without being frequently interrupted by exceptions caused by data races). To address these issues, we first extend from prior approaches to avoid data race caused exceptions under a strong memory model. We introduce *waiting* at program points that detect conflicts, in an effort to tolerate the conflict and avoid raising an exception. An evaluation on benchmarked versions of large, real Java programs shows that our approach improves availability substantially, compared with their counterparts that do not wait at conflicts. However, the existing strong memory model is a strong model that may inherently limit availability and/or cost and complexity, whether implemented purely in software or with hardware support.

We thus introduce an alternative memory model called *SIx* based on *snapshot isolation* of regions (SI). SIx ensures region isolation and provides significantly stronger guarantees for racy executions than existing weak memory models. To provide SIx, we introduce an approach called *Snappy* that correctly "tolerates" *read–write conflicts* by deferring handling of the conflicts until region end. Our evaluation shows that Snappy can provide better availability than prior approaches, although it incurs high overhead in order to detect conflicts precisely.

Our final insight is that Snappy can detect read–write conflicts *imprecisely* (false positives but no false negatives) without jeopardizing support for SIx. We leverage this idea to introduce a new approach, called *Snappy-I*, that represents variables' last reader information imprecisely, enabling lower runtime costs and complexity than the other approaches.

However, Snappy-I's reduced precision gives up most of the availability gains that Snappy-P provides over the approaches that provide RSx.

Overall, our exploration and evaluation of the design space exposes new directions for supporting strong memory models with both strong semantics and high availability while still allowing compilers and hardware to perform optimizations within code regions. Furthermore, our design for enforcing SIx could enable simpler hardware designs, just as Snappy-I yields a simpler software design than its competitors.

1.4 Dissertation Contributions

This dissertation studies concurrency control mechanisms in programming models, runtime support, and memory consistency models. The following paragraphs describe the organization and contributions of this dissertation.

Chapter 2 introduces the necessary definitions and concepts used throughout the dissertation.

Chapter 3 introduces how relaxed tracking (RT) improves the performance of runtime support by hiding coordination latency. RT also solves problems that exist in LarkTM, and in particular, the RT-based STM is an extension of LarkTM. The contribution of this part includes:

- a novel dependence-tracking mechanism (relaxed dependence tracking) that improves scalability by hiding synchronization costs;
- relaxed dependence tracking preserves both program semantics and runtime support guarantees;

- implementations of (i) relaxed dependence tracking and (ii) a RT-based recorder and a RT-based STM; and

- an evaluation between relaxed dependence tracking and strict dependence tracking to show that relaxed dependence tracking is able to improve performance of runtime support, especially for applications with high-contention.

Chapter 4 introduces the details of the low-overhead, scalable software transactional memory system with strong semantics. The contributions of this part include:

- a novel STM called LarkTM that (i) adds low overhead by making non-conflicting accesses fast, and synchronization free, (ii) provides strong progress guarantees, and (iii) supports strong semantics efficiently;

- a novel approach for integrating LarkTM's concurrency control mechanism with an existing STM concurrency control mechanism that has different tradeoffs, yielding basic and adaptive STM versions (LarkTM-O and LarkTM-S);

- implementations of (i) LarkTM-O and LarkTM-S and (ii) two high-performance STMs from prior work; and

- an evaluation on transactional benchmarks that shows that LarkTM-O and LarkTM-S achieve low overhead and good scalability, thus outperforming existing high-performance STMs.

Chapter 5 introduces a novel STM design, called RT-based STM, that combine the features of both LarkTM and RT in order to support a low-overhead and strongly atomic STM with better scalability than the ST-based STM. Most STMs employ either entirely lazy or

entirely eager concurrency control (e.g., [141, 49, 56, 164]). Some STMs combine lazy and eager mechanisms, by using eager concurrency control for *writes* and lazy validation for *reads* [124, 76]. RT-based STM combines eager and lazy concurrency control in a novel way, using eager and lazy concurrency control for non-conflicting and conflicting accesses, respectively.

Chapter 6 introduces novel and strong memory models that provide strong consistency guarantees even for ill-synchronization programs with data races. Furthermore, to the best of our knowledge, it is the first to consider the problem of *availability* for memory consistency models that throw consistency exceptions. This part makes the following contributions:

- approaches for providing RSx that differ from prior work in that they have better availability;
- a new memory consistency model called *SIx* that is weaker than RSx but still ensures isolation of regions for racy executions;
- an approach that enforces SIx, and two designs with differing tradeoffs; and
- evaluations of availability, runtime performance, scalability, and other characteristics of our approaches and existing approaches that provide RSx and SIx.

Chapter 7 presents related work for the work we have introduced. Finally, Chapter 8 concludes the dissertation and describes directions for future work.

1.5 Meaning and Impact

Our techniques use novel insights to build efficient runtime support to get reliable and scalable parallelism on shared-memory systems and are more efficient than prior approaches. Our dependence tracking technique effectively reduces the coordination latency by relaxing the condition that dependence tracking needs to track dependences soundly. Our transactional memory design and implementation rely on innovative ways to detect and resolve conflicts between both *transactional and transactional* code and *transactional and non-transactional* code. Our strong memory model work addresses the "availability" issue of exception-based strong memory models and demonstrates techniques that can significantly improve the availability of two strong memory models. Our implementation developed in this dissertation is publicly available, and following up work has been built upon on these implementations.

We believe that algorithms, tools, and systems developed in this dissertation will help more people write efficient concurrent programs and take advantage of the power of multicore parallel hardware. It points to a future where (1) developers build more efficient runtime support with relaxed dependence tracking; (2) more users embrace transactional memory as a simple programming synchronization model for writing concurrent programs because of its low-overhead, good scalability, and strong semantics; and (3) both users and developers can more reliably reason about concurrent programs with a strong memory model that has well-defined behaviors even when there are data races. Overall, as concurrent programming becomes easier and more efficient, more software can benefit from ever-increasing parallel hardware.

Chapter 2: Preliminaries and Background

This chapter presents the definitions, concepts, and base approaches that will be used throughout the dissertation, which we have also published separately [163, 164, 162]. Individual chapters have additional definitions and notations that are specific to that chapter.

Section 2.1 introduces the problem of tracking cross-thread memory dependences in runtime support. Section 2.2 describes concepts related to transactional memory (TM). Section 2.3 introduces problems and challenges related to memory consistency models.

2.1 Capturing Cross-Thread Dependences

Capturing cross-thread dependences is useful for checking or enforcing concurrency runtime properties, such as atomicity, ordering, and determinism. More specifically, it usually means one of the following two things:

Tracking cross-thread dependences. Examples of runtime support[1] that need to soundly (and sometimes precisely) track cross-thread dependences are data race detectors (e.g., [63, 58]), atomicity checkers (e.g., [64, 16]), and dependence recorders for deterministic record & replay (e.g., [89, 28]).

[1] Although data race detectors and atomicity checkers are often called "dynamic analyses," here we classify them generally as "runtime support."

Controlling cross-thread dependences. Examples of runtime support that need to soundly control cross-thread dependences are transactional memory (e.g., [79, 73]), support for stronger memory models (e.g., [114, 131]), and deterministic execution (e.g., [112, 14]).

For data-race-free (DRF) executions, capturing cross-thread dependences requires only tracking or controlling *synchronization* operations, since DRF0-based memory models including the Java and C++ memory models guarantee atomicity of synchronization-free regions for DRF executions [100, 23, 4, 6].

However, concurrent programs routinely have intentional and unintentional data races, which languages and program analyses have failed to eliminate despite substantial effort (e.g., [108, 63, 29, 89]). Thus, to capture cross-thread dependences, runtime support adds instrumentation at each access that might be involved in a data race. (Even after applying sound static data race detection as a filter, many accesses cannot be proven to be DRF [150, 39, 58, 89].)

Furthermore, to ensure that dependences are captured soundly, runtime support needs to ensure that an access and its instrumentation execute together atomically, a property we call *instrumentation–access* atomicity. To preserve instrumentation–access atomicity, runtime support often synchronizes on a lock associated with each object[2] (e.g., represented with an extra word in the object's header). The runtime support's instrumentation acquires the lock for reading and/or writing the object. An implementation typically relies on atomic operations and memory fences, which introduce remote cache misses and serialize in-flight instructions (e.g., [135, 63, 88, 64, 89]).

[2]The dissertation uses the term "object" to refer to any unit of shared memory.

Therefore, capturing cross-thread dependences at almost every access to potential shared memory location adds high synchronization overhead. Record & replay system, for example, usually suffers from poor performance because of this high cost to capture dependences during record in order to enforce that order during replay.

We note that some approaches have sidestepped these challenges but incur other limitations. For example, record & replay can avoid tracking dependences by relying on replication and speculation, but its performance relies on extra available cores [148]. Some analyses, notably data race detection, need not preserve instrumentation–access atomicity, but still require instrumentation atomicity, which ends up incurring similarly high costs [63, 103].

2.1.1 Biased Reader–Writer Locking

As tracking cross-thread dependences efficiently is the fundamental challenge in solving many concurrency issues (e.g., it is applicable to both record & replay, software transactional memory, data race detection, and atomicity violation detection), my coauthors and I have investigated how to track cross-thread dependences *efficiently*. Prior work introduces so-called *biased locks*, in which each lock is "owned" by one thread, which can acquire the lock without using an atomic operation [123, 82, 126, 28, 149, 80, 129, 32]. However, another thread that wants to acquire a lock owned by another thread must *coordinate* with the owner thread, in order to establish a roundtrip happens-before relationship with the owner thread, which might otherwise continue to access the object that the lock guards without performing synchronization. Biased locking offers low overhead for tracking cross-thread dependences. In contrast, tracking dependences using traditional reader–writer locks often slows down programs by 3–4X [35].

In this dissertation, we describe a low-overhead design of biased locks called *biased reader–writer locks* that is most closely based on prior work from my coauthors and me called *Octet* [28]. We choose Octet since it supports read-shared pattern better than existing work and it achieves lower overhead than most existing approaches. Furthermore, our dependence recorder and software transactional memory are extended from biased reader–writer locks. With biased reader–writer locking, each object's lock can have any of the following states:

- $\mathsf{WrEx}_{\mathsf{T}}$: write-exclusive for thread T.
- $\mathsf{RdEx}_{\mathsf{T}}$: read-exclusive for T.
- $\mathsf{RdSh}_{\mathsf{c}}$: read-shared for all threads, subject to a counter c that helps detect dependences from the last write soundly [28].

When thread T allocates an object, the object's lock is acquired in the $\mathsf{WrEx}_{\mathsf{T}}$ state. Table 2.1 shows state transitions for locks for every possible initial state of an object and access. The *Same state* rows correspond to cases where the lock is acquired in the needed state, and no synchronization is needed. In *Upgrading* transitions, accesses allowed under the old state are allowed under the new state, so coordination is unnecessary; an atomic operation is needed in case two threads both try to change the state at the same time.

Fence transitions detect cases where a thread has not yet read an object in the $\mathsf{RdSh}_{\mathsf{c}}$ state since the last write to the object, ensuring detection of transitive happens-before relationship from the last write to the current read. Prior work provides more details on fence transitions and the read-shared counter c [28].

Conflicting transitions handle cases where the thread(s) that currently hold an object's lock can perform accesses that would conflict with the pending access. The new thread

Code path(s)	Transition type	Old state	Program access	New state	Sync. needed
Fast	Same state	$\mathsf{WrEx_T}$	R/W by T	Same	None
		$\mathsf{RdEx_T}$	R by T	Same	
		$\mathsf{RdSh_c}$	R by T *	Same	
Fast & slow	Upgrading	$\mathsf{RdEx_T}$	W by T	$\mathsf{WrEx_T}$	Atomic op.
		$\mathsf{RdEx_{T1}}$	R by T2	$\mathsf{RdSh_c}$	
	Fence	$\mathsf{RdSh_c}$	R by T *	$\mathsf{RdSh_c}$	Mem. fence
	Conflicting	$\mathsf{WrEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	Roundtrip coord.
		$\mathsf{WrEx_{T1}}$	R by T2	$\mathsf{RdEx_{T2}}$	
		$\mathsf{RdEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	
		$\mathsf{RdSh_c}$	W by T	$\mathsf{WrEx_T}$	

Table 2.1: State transitions for biased reader–writer locks. * Acquiring a read lock on a $\mathsf{RdSh_c}$ object by T triggers a fence transition if and only if per-thread counter T.rdShCount $<$ c. Fence transitions update T.rdShCount to c.

initiates a roundtrip *coordination protocol* to ensure that the other thread(s) relinquish the lock safely. The new thread must wait for the other thread(s) to respond.

Figure 2.1 shows the instrumentation that the compiler adds at each program load and store; the instrumentation "acquires" a write or read lock on the accessed object's lock. If a thread already owns the lock for an object, the instrumentation takes the synchronization-free, write-free *fast path*. Otherwise, the instrumentation executes the *slow path*, which changes the state and handles potential cross-thread dependences, as described next.

2.1.1.1 Conflicts Require Coordination

When a thread needs to acquire a read or write lock that it does not already own (e.g., at a read by T2 to an object whose lock is in $\mathsf{WrEx_{T1}}$ state), it must *coordinate* with the owner thread(s), so they can acknowledge the ownership transfer and cease unsynchronized accesses to the object.

```
1   if (o.state != WrEx_T) {
2     slowPath(o); /* acquire o.state for write */
3   }
4   o.f = ...;  // program store to the object field o.f

5   if (o.state != WrEx_T &&
6       o.state != RdEx_T &&
7       (o.state != RdSh_c || T.rdShCount < c)) {
8     slowPath(o); /* acquire o.state for read */
9   }
10  ... = o.f; // program load to the object field o.f
```

Figure 2.1: Pseudocode for biased reader–writer locking's instrumentation fast path. T is the executing thread.

Handling conflicting transitions with coordination. In a conflicting transition, suppose a thread, called the *requesting thread*, reqT, wants to acquire an object's lock held in a conflicting state by other thread(s); each of these other thread(s) is a *responding thread*, respT. If the object's lock is in $\mathsf{WrEx}_{\mathsf{respT}}$ or $\mathsf{RdEx}_{\mathsf{respT}}$ state, then there is one responding thread, respT. If the object's lock is in RdSh state, then all other threads are responding threads, and reqT coordinates with each responding thread separately. For simplicity of exposition, we describe the case of a single responding thread respT.

The pseudocode in Figure 2.2 shows the slow path.[3] First, reqT atomically changes the state of the object's lock to an *intermediate* state, $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ (depending on whether a read or write lock is needed), with a CAS (line 17).[4] The intermediate state simplifies the protocol by allowing only one thread at a time to perform a conflicting transition on an object's lock. If there is another thread that has already changed the object to an intermediate state, reqT waits for the other thread to finish coordination (lines 17–19).

[3]The pseudocode omits memory fences required by the implementation.

[4]The atomic operation CAS(addr, oldVal, newVal) attempts to update addr from oldVal to newVal, returning true on success.

```
11 slowPath(o) {
12   state = o.state;
13   // Handle non−conflicting state transitions :
14   if (state == ...) { ...; return; }
15   // Coordination for conflicting transitions :
16   while (state == RdEx*^Int || state == WrEx*^Int
17          || !CAS(&o.state, state, RdExT^Int)) { // or WrExT^Int
18     state = o.state;  // re-read state
19   }
20   coordinate(getOwner(state));
21   o.state = RdExT; // or WrExT
22 }
23 coordinate(remoteT) {
24   response = sendCoordinationRequest(remoteT);
25   while (!response) {
26     response = status(remoteT);
27   }
28 }
```

Figure 2.2: Pseudocode for biased reader–writer locking's slow path. T is the executing thread.

After reqT successfully changes the object's lock state to $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$, it coordinates with respT (line 20) to ensure that reqT's state change does not interfere with respT's instrumentation–access atomicity. respT participates in coordination only when it is at a *safe point*: a program point that is definitely *not* in the middle of instrumentation or its corresponding access—hence preserving instrumentation–access atomicity. Managed language VMs already place safe points at periodic points in compiled code (e.g., method entries and exits and loop back edges) for profiling and timely yielding for parallel garbage collection. *Blocking* operations such as waiting to acquire a program lock or waiting for I/O are also safe points.

If respT is at a *blocking safe point*, reqT makes an *implicit* request to respT (at line 24) by atomically updating respT's status, which respT will see when it leaves the blocking state. The helper method sendCoordinationRequest() returns true if and only if it performs

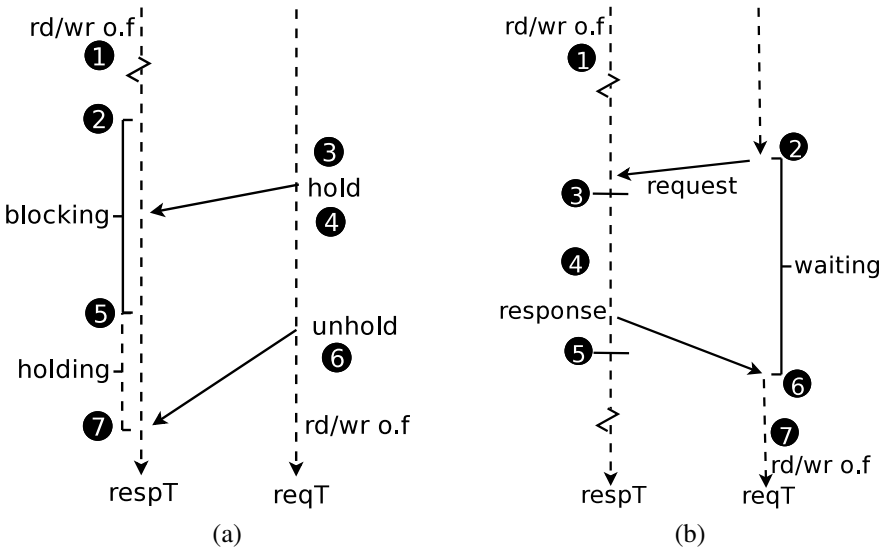


Figure 2.3: The biased reader–writer locking coordination protocol. **(a) Implicit request**: (1) respT accessed o previously. (2) respT enters a blocked state before performing some blocking operation. (3) reqT wants to access o. It changes o's lock's state to $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$. (4) reqT performs runtime-support-specific actions while keeping respT in a "blocked and held" state. (5) respT finishes blocking but waits until hold(s) have been removed. (6) reqT removes the hold on respT and changes o's lock's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$ and proceeds to access o. (7) respT leaves the "blocked and held" state. **(b) Explicit request**: (1) respT accessed an object o previously. (2) reqT wants to access o. It changes o's lock to $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$, and enters a blocked state, waiting for respT's response. (3) respT reaches a safe point. (4) respT performs runtime-support-specific actions and then responds. (5) respT leaves the safe point. (6) reqT sees the response. (7) reqT changes o's lock's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$ and proceeds to access o.

an implicit request. Otherwise, reqT sends an *explicit* request to respT: reqT sends a request to respT by adding a request to respT's *request queue*, and then must wait (lines 25–27) for respT to reach a safe point to respond. While reqT is performing coordination (lines 16–20, including the body of coordinate), it is considered to be at a blocking safe point (mechanism not shown in the pseudocode), to allow other threads to perform implicit requests with reqT acting as a *responding* thread, thus avoiding deadlock. Figure 2.3 illustrates how coordination works when using an implicit or an explicit request.

Explicit protocol. If respT is *not* at a blocking operation, reqT performs the *explicit* protocol as shown in Figure 2.3(a). reqT requests a response from respT by adding itself to respT's *request queue*. respT handles the request at a safe point. respT performs conflict detection and resolution while reqT is stopped, ensuring safety. Then respT responds to reqT. Once reqT receives the response, it ensures that respT will "see" that the object's lock's state has changed. During the *explicit* protocol, while reqT waits for a response, it enters the blocked state so that it can act as a *responding* thread for other threads performing the implicit protocol, thus avoiding deadlock.

Implicit protocol. If respT is performing a blocking operation, then reqT performs the *implicit* protocol as shown in Figure 2.3(b). reqT atomically "places a hold" on respT by putting it in a "blocked and held" state. If respT finishes its blocking operation, it will wait for the hold to be released before continuing execution, allowing reqT to read and potentially modify respT's state safely. Multiple requesting threads can place a hold on respT, so the hold state is actually a counter that indicates the number of holds placed on respT. After reqT performs conflict detection and resolution, it removes the hold by decrementing respT's hold counter. Once respT's hold counter drops to zero, it can proceed.

After either protocol completes, reqT changes the object's lock's state to the new state ($\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$)—unless reqT aborts, in which case the protocol reverts the lock to its old state (Section 4.1.3). Finally, reqT changes the state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$ (line 21) and proceeds with its access to the field.

2.1.1.2 Coordination Cost

Coordination can slow programs substantially, even for programs that perform relatively few conflicting accesses (e.g., 0.1–1% of accesses triggering coordination) [35].

The following table reports the average cost of coordination using explicit versus implicit requests, compared with the cost of instrumentation that does not change the lock's state [163]:

	Same state	Implicit request	Explicit request
CPU cycles	47	360	9,200

Coordination is substantially more expensive than a same-state check because coordination requires several memory accesses, including atomic operations and memory fences. On average, coordination using an explicit request is more than an order of magnitude more costly than using an implicit request, since an explicit request incurs significant latency waiting for roundtrip communication. Our work in Chapter 3 thus focuses on optimizing coordination that uses explicit requests. Our STM in Chapter 4 addresses this issue by using an adaptive approach to let objects that causes many conflicting accesses to use a concurrency control mechanism that incurs less overhead to transfer lock ownership.

2.2 Transactional Programming

As Section 1.1 mentions, data races can have harmful or even disastrous effects. It is therefore imperative to make a concurrent program data-race-free, and the program should be written in a way that can be be verified with confidence during development and testing phase that it does not contain data races. A common approach to prevent data races on shared data is to use locks (or any kind of mutual exclusion). However, locks are hard to use. Taking too few locks can compromise data integrity. Taking too many locks can cause deadlocks or hampers concurrency. Programmers need to identify and remember the connection between a lock and the data it protects, which can cause themselves or other people to take the wrong locks or in wrong order. Locks make error recovery difficult as it can leave the system in an inconsistent state. Worst of all, third party libraries written

using locks cannot be easily composed, making them hard to retain scalability and avoid deadlock and data races.

A different synchronization programming model is transactional memory (TM). TM allows a thread to perform a sequence of instructions as if a single instruction, and guarantees that the execution of a sequence of transactons is serializable. Under this programming model, the hazard of data races can be effectively reduced as it allow programmers to move up from low level locks to more abstract concepts at a higher level. The underlying TM system optimizes the actual concurrency of the execution by overlapping transactions wherever it is possible without violating serializability. To highlight the advantages of TM, Grossman makes an interesting analogy between TM and garbage collection that TM would make writing concurrent programs easier as garbage collection made memory management easier [70].

This section gives background on transactional memory and introduces basic concepts that are used later in the text. Harris et al. provide a detailed overview of TM [75].

2.2.1 Design

Programming Model and Interface From a programmer's point of view, the high-level interface provided by TM is an ***atomic block***. Programmers can use an atomic block to denote a sequence of operations that should be executed atomically, i.e., all operations inside of the block appear to execute as a single atomic operation. For example, the following code snippet is an atomic block that has two statements in it. These two statements are executed as one indivisible operation.

```
atomic {
  statement_1;
  statement_2;
}
```

Next, we introduce design components that are common for building TM.

Versioning. To enable transactions to execute speculatively, TM systems provide *versioning* of transactional stores to potentially shared memory. *Lazy* versioning *buffers* stored memory values. In software transactional memory (STM) systems, this buffer is called a *redo log*. The TM system writes the buffered values to memory if and when the transaction commits. In STM systems, lazy versioning adds overhead because each load needs to check the redo log for a recent store to the same location.

Eager versioning buffers *old* values of modified memory locations in an *undo log*. If and when a transaction aborts, the TM system writes these old values back to memory. Although eager versioning typically provides better performance, most STM systems do not use it because it is incompatible with lazy conflict detection and resolution.

Read/write sets To support precise conflict detection, TMs record the addresses of shared memory read and written during each transaction in *read/write sets* [107]. Hash-table-based read/write sets provide precise conflict detection but slow transactional reads and writes; Bloom filters [20] are faster but cause spurious conflicts due to imprecision [166]. Alternatively, each transaction can acquire read and write locks on shared memory and release these locks when the transaction aborts or commits [135]. Another transaction or non-transactional access conflicts if it tries to acquire a conflicting lock.

Conflict detection. *Concurrency control* consists of conflict detection and conflict resolution. *Conflict detection* detects read–write and write–write conflicts between two overlapping transactions—or between a transaction and a non-transactional access in strongly atomic STMs. Conflict detection can be eager (also called pessimistic) or lazy (also called

optimistic). *Eager* conflict detection identifies conflicts as soon as a thread tries to perform an access that conflicts with an access in other thread(s)' in-progress transaction. *Lazy* conflict detection occurs later, typically when a transaction tries to commit. Lazy conflict detection typically improves STM performance by reducing how much synchronized instrumentation is needed, but it is incompatible with eager versioning.

Conflict resolution. *Conflict resolution* resolves detected conflicts, typically by aborting one or more transactions. Aborting a transaction involves rolling back its speculative state (in the case of eager versioning) and returning control to the start of the execution. *Eager* (pessimistic) conflict resolution, which requires eager conflict detection, resolves conflicts as soon as they are detected. *Lazy* (optimistic) conflict resolution resolves conflicts later, typically when a transaction tries to commit. Eager versioning requires eager conflict resolution: a transaction requires exclusive access to memory locations if it is going to directly update them, or else a transaction can execute unserializable code indefinitely (so-called "zombie" transactions).

Contention management. *Contention management* decides which transaction to abort (or in some cases, delay) when a conflict is detected. There has been significant work on contention management policies (e.g., [141]). Contention management is particularly important for high-contention workloads, for which simple policies often perform poorly.

Overall, the design space for TM is large. In Chapter 4, we show how we reduce the overhead of conflict detection and read/write sets in STM to get decent overhead and use an adaptive approach to get decent scalability.

2.2.2 Performance and Semantics

Existing STMs have been facing challenges mainly from performance and semantics. Unlike hardware-supported TM (Section 7.2), STM slows programs significantly due to the overhead introduced by supporting conflict detection and rollback. STM slows programs further if it provides strong behavioral guarantees for data races between transactions and non-transactional accesses.

Performance. Some researchers doubt whether STM is or can be made practical enough [160, 37, 55]. One way to measure STM's success is whether an STM system can provide enough parallelism to achieve a speed up over single-threaded execution—sometimes a difficult feat because of STM's huge single-threaded slowdown. The very fact that this measure has been applied shows how much overhead existing STM systems add to single-threaded performance. Overall, many researchers are pessimistic about whether STM can be made efficient because of its performance [37, 160].

Recent high-performance STMs typically use lazy versioning and concurrency control [49, 56, 54, 55, 143, 113] (although SwissTM detects write–write conflicts eagerly [56, 55]). Lazy mechanisms can make contention management (i.e., choosing which transaction to abort) to be more effective by deferring decisions until commit time [141]. Lazy mechanisms can also provide lower synchronization costs—a key STM cost [160]—than eager mechanisms: instead of requiring synchronization for every distinct accessed object in a transaction, some lazy STMs use a global clock or global metadata to perform as little as one synchronization operation for each transaction [49, 143, 113]. For example, NOrec acquires a single global lock to perform buffered writes [49]. Trading per-access costs for per-transaction costs is not well suited to providing strong atomicity, since each

non-transactional access essentially executes as its own tiny transaction. By using lazy mechanisms for reads, STM can avoid atomic operations at reads, although memory fences are still needed (e.g., [49, 135]).

Some STMs have used eager concurrency control for *writes*, but lazy concurrency control for *reads* (so-called "invisible reads") in order to avoid synchronization costs at reads [124, 135, 76, 128]. Notably, we implement and compare against an STM that we call *IntelSTM*, Shpeisman et al.'s strongly atomic version [135] of McRT-STM [124] (Chapter 4). IntelSTM and other mixed-mode STMs detect write–write and write–read conflicts eagerly but detect read–write conflicts lazily by logging reads and validating them later.

Alternatively, other STMs, including LarkTM introduced in this dissertation, use eager versioning and eager conflict detection and resolution. Eager STMs have the potential to be faster than lazy STMs, e.g., lazy versioning adds extra overhead to stores (to buffer and replay them) and loads (to look up stored values); even state-of-the-art lazy STMs add overhead that slows execution by several times (e.g., [49]). Furthermore, eager STMs can provide strong atomicity if they acquire locks on non-transactional accesses. However, overcoming eager mechanisms' high synchronization costs has proven challenging.

Transactional semantics. Most existing STMs provide *weak atomicity* semantics: they detect and resolve conflicts between two transactions but not between a non-transactional access and a transaction. Researchers generally agree that STMs must provide at least *single lock atomicity* (SLA) semantics (or a relaxed variant such as asymmetric lock atomicity [104]): behavior should be the same as if each transaction were replaced with a critical section acquiring a global lock. SLA (and its variants, for the most part) provide safety for

so-called *privatization* and *publication* patterns, in which data-race-free accesses to an object occur both in and out of transactions [135, 106, 1]. Weakly atomic STMs can provide privatization safety using various techniques that can hurt scalability [160], such as by committing transactions in the same order that they started [104, 154, 142], or by committing writes using a global lock [49].

A stronger memory model than SLA is *strong atomicity* (also called strong isolation), which provides atomicity of transactions with respect to non-transactional accesses. Strong atomicity guarantees not only privatization and publication safety, but it always executes transactions atomically, even when there are data races between transactions and non-transactional accesses. Some researchers have argued that strong atomicity is not worth its substantial cost: strong atomicity is only stronger than SLA for programs that have data races between transactions and non-transactional accesses [46, 48]. Still, programmers often fail to eliminate data races due to the challenge of using locks and shared memory. Adve and Boehm have argued that stronger behavior guarantees are needed for (non-transactional) racy programs [4]. Hardware TM typically provides strong atomicity, making it an appealing memory model for an STM used in hybrid software–hardware TM.

Prior work reduces the cost of strongly atomic STM by using static and dynamic analyses to eliminate instrumentation at non-transactional accesses. Shpeisman et al. use whole-program static analysis and dynamic thread escape analysis to identify thread-local accesses that cannot conflict with a transaction and thus do not need expensive instrumentation [135]. We call this work IntelSTM. IntelSTM is the strong atomicity version of McRT-STM. For transactional accesses, IntelSTM handles them the same way as McRT-STM does. IntelSTM requires each non-transactional write to acquire a lock before doing updates, and requires each non-transactional read to check whether its version number

has been changed before and after itself. This approach could add high instrumentation overhead. IntelSTM therefore relies on whole program analysis to eliminate redundant instrumentation. While the authors report relatively low overheads, the evaluation uses the notoriously simple SPECjvm98 benchmarks.[5] It is unsurprising that whole-program static analysis and dynamic thread escape analysis work well, especially since all but two of the benchmarks are single-threaded.

Progress guarantees. STMs can suffer from *livelock*: two or more threads' transactions repeatedly cause each other to abort and retry. STMs that use lazy concurrency control for both reads and writes can help to guarantee freedom from livelock. For example, NOrec can always commit at least one transaction among a set of concurrent transactions [49]. (Lazy mechanisms provide two additional benefits in prior work. First, they help to provide sandboxing guarantees for unsafe languages such as C and C++ [47]. In contrast, our design targets safe languages and does not require sandboxing; Section 4.2. Second, for high-contention workloads, lazy concurrency control helps make *contention management*, i.e., choosing which conflicting transaction to abort, more effective by deferring decisions until commit time [141].)

Although fully lazy STMs can help to guarantee livelock freedom, they cannot generally guarantee *starvation* freedom: not only will at least one thread's transaction eventually commit, but every thread's transaction will eventually commit. STMs that use eager concurrency control for both reads and writes, including our LarkTM, can guarantee not only livelock freedom but also starvation freedom, as long as they provide support for aborting

[5]The more-realistic DaCapo Benchmarks [18] were probably not yet available when Shpeisman et al. conducted their evaluation.

either thread involved in a conflict (since this flexibility enables age-based contention management; Section 4.1.3) [71]. (An interesting related design is *InvalSTM*, which uses fully *lazy* concurrency control and allows a thread to abort *another* thread's transaction [69].)

In contrast, STMs such as IntelSTM that mix lazy and eager concurrency control struggle to guarantee livelock freedom: since any transaction that fails read validation *must* abort, all running transactions can repeatedly fail read validation and abort [71, 135].

2.3 Memory Consistency Models

This section introduces how existing memory models and how they can affect performance and reliability of concurrent programming.

DRF0. Modern shared-memory languages such as Java and C++ provide variants of the *DFR0* memory model, introduced by Adve and Hill in 1990 [6, 23, 100]. DRF0 (and its variants) provides a strong guarantee for well-synchronized, or *data-race-free*, executions: *serializability of synchronization-free regions* (SFRs) [4, 98].[6] An SFR is a dynamic sequence of executed instructions bounded by synchronization operations (e.g., lock acquires and releases) with no intervening synchronization operations. An execution is region serializable if it is equivalent to some serial execution of regions (i.e., some global order of non-interleaved regions).

For executions with data races, DRF0 provides weak or no behavior guarantees [21, 22, 4, 24, 25]. C++'s memory model gives undefined semantics for data races [23]. The Java memory model (JMM), on the other hand, provides well-defined but weak semantics for racy executions, in an effort to preserve memory and type safety [100]. However,

[6]DRF0 also provides *sequential consistency* (SC) [86] for DRF0 executions. SFR serializability implies SC.

as researchers later discovered, the JMM precludes common Java virtual machine (JVM) compiler optimizations [133]. The state of practice is that JVMs perform optimizations that violate the JMM [25]. According to Adve and Boehm, "The inability to define reasonable semantics for programs with data races is not just a theoretical shortcoming, but a fundamental hole in the foundation of our languages and systems" [4].

Despite much effort by researchers and practitioners, data races are difficult to avoid, detect, fix, and eliminate (e.g., [132, 29, 2, 109, 108, 151, 60, 42, 58, 161, 119, 111, 41, 61, 125, 149, 26, 63, 17, 81]). Data races often manifest only under certain environments, inputs, and thread interleavings, allowing them to go undetected [165, 146, 68, 97]. Data races thus occur unexpectedly in production systems, sometimes with severe consequences [91, 146, 117]. They often indicate concurrency bugs such as atomicity, order, and sequential consistency violations [97]. Thus, systems that execute with weak or undefined semantics due to data races are not just a problem in theory but in practice.

In spite of the shortcomings of DRF0-based memory models, languages and systems continue to use them in order to maximize performance. DRF0 allows compilers and hardware to perform uninhibited *intra-thread* optimizations, as long as optimizations do not cross synchronization operations. Any attempt to provide stronger consistency must consider the impact of restricting optimizations.

Sequential consistency. Much work has focused on providing *sequential consistency* (SC)[7] as the memory consistency model [102, 138, 5, 4, 67, 92, 120, 93, 134, 144]. Enforcing *end-to-end* SC (i.e., SC with respect to the original program) requires restricting optimizations by both the compiler and hardware. (In contrast, providing SC in the compiler or hardware alone does not provide end-to-end SC.)

[7]Under SC, operations appear to interleave in some order that conforms to program order [86].

Although SC is certainly stronger than the undefined or weak behaviors that DRF0 provides for racy executions, it is not a particularly strong model. Programmers tend to think in terms of operations that are larger than individual memory accesses, expecting a multi-access operation such as `x++` or `buffer[index++] = 42` to execute atomically (regardless of the actual memory model). Adve and Boehm argue that "programmers do not reason about correctness of parallel code in terms of interleavings of individual memory accesses" and that SC "does not prevent common sources of concurrency bugs . . . " [4].

Region serializability. An alternative to DRF0 and SC is memory consistency based on *region serializability*. Notably, region serializability of SFRs, which we abbreviate as *RS*, extends the same guarantees to *all* executions that DRF0 provides for race-free executions only. Under RS, an execution is equivalent to some execution in which SFRs (executed sequences of instructions that do not contain synchronization operations) appear to execute serially, i.e., without interruption by other threads [98, 17, 114]. RS is appealing because (1) it provides the same strong guarantees for all executions that DRF0 already provides but *only* for race-free executions, and (2) it does not restrict compiler and hardware optimizations, which already respect synchronization operations as region boundaries.

However, enforcing RS seems inherently problematic due to supporting unbounded speculative execution (Section 7.3). Researchers have thus introduced a memory consistency model that we call *RSx* that treats data races as errors, potentially throwing a *consistency exception* for a data race, but otherwise ensuring RS [98, 17]. (Furthermore, other approaches treat some or all data races as errors [58, 156, 130].)

RSx either (1) ensures RS or (2) generates a *consistency exception*—but only if there exists a data race. RSx allows for some flexibility: an approach does *not* need to incur the cost and complexity of soundly and precisely detecting all data races nor all RS violations.

Under RSx, data races are errors, just like buffer overflows and other memory errors in memory- and type-safe languages such as Java. In contrast, under DRF0, data races lead to undefined semantics, like buffer overflows in unsafe languages such as C++. In an RSx-by-default world, programmers will need to identify and eliminate most or all data races that cause consistency exceptions during testing, just as programmers now debug in response to exceptions from buffer overflows and other memory errors in safe languages. Nonetheless, even well-tested software may contain unknown data races that may manifest only under certain production environments, inputs, or thread interleavings [146, 117, 91], leading to unexpected consistency exceptions. Unexpected exceptions hurt the *availability* (this dissertation's term for exception freedom) of production systems.

2.4 Implementation and Methodology

This section describes implementations and experimental methodology that are common for all the work introduced in this dissertation.

2.4.1 Implementations

All implementations described in this dissertation are done in Jikes RVM, a high-performance virtual machine [10]. As of October 2015, Jikes RVM performs competitively with OpenJDK on DaCapo benchmark suites: If we exclude an outlier benchmark pjbb2005, Jikes RVM is 20% slower (AMD) and 7% faster (Intel) than OpenJDK [17]. Jikes RVM therefore performs competitively with modern commercial JVMs and our performance measurements are relative to a good baseline.

All our implementations either are or will be made publicly available on the Jikes RVM Research Archive:

- Four STMs: LarkTM-O, LarkTM-S, NOrec [49], and IntelSTM [135] as a patch against Jikes RVM 3.1.3 (Chapter 4).

- Relaxed dependence tracking (RT), RT-based recorder and RT-based STM, as a patch against Jikes RVM 3.1.3 (Chapter 3). Our RT implementation builds on our publicly available strict dependence tracking (ST) implementation called *Octet* [28]. Our RT-based recorder builds on the publicly available ST-based recorder [28, 27]. Our RT-based STM (Chapter 5) builds on our publicly available ST-based STM called *LarkTM* [164]. Our implementations reuse features of the ST-based implementations as much as possible.

- Snappy will be available once the work has been published (Chapter 6). The source code will be a patch against Jikes RVM 3.1.3 and includes FastRCD-A, Valor-A, Snappy-P, and Snappy-I. Our FastRCD-A and Valor-A implementations build on our publicly available FastRCD and Valor [17]. We remove redundant memory accesses in FastRCD's read instrumentation without affecting its functionality. Our Snappy-P and Snappy-I implementations instrument the exactly same memory accesses as FastRCD. All four implementations demarcate synchronization-free-region in the same way.

2.4.2 Methodology

Jikes RVM uses *just-in-time* (JIT) compilation and automatic memory management. This section describes how our implementations use these features and common methodology in our experiments.

Compilers. Jikes RVM has two compilers to produce machine code for each application method at runtime. The first one is a *baseline* compiler that takes Java bytecode and generates machine code when the application first executes a method. The second one is a Just-in-Time (JIT) compiler that recompile the bytecode of a method to machine code on the fly at runtime. Jikes RVM's *adaptive system* uses timer-based sampling to identify hot methods. When a method executes many times and becomes hot, the VM launches the JIT compiler to recompile the method at successively higher optimization levels. The JIT compiler uses *intermediate representation* (IR) and performs optimizations such as inlining, constant propagation, and register allocation. The implementations in this dissertation modify both compilers to add instrumentation to application code.

Memory Management. Jikes RVM supports a variety of garbage collectors (GC). By default, we use the high-performance generational Immix GC [19]. It allocates new objects into a *nursery space* and then periodically moves surviving objects to an Immix *mature space*. The immix space uses mark-sweep collection and minimizes fragmentation to maximize freed blocks. Furthermore, heap size can affect garbage collection frequency and workload. We let GC decide and adjust the size of the heap automatically for each benchmark.

Execution. To account for run-to-run variability due to the JVM's nondeterministic compilation and execution, each of our performance results executes 25 trials. We plot the median, to minimize the effects of possible machine noise. We also show the mean, as the center of 95% confidence intervals.

Benchmarks. To evaluate overhead and scalability of STMs, we use the transactional *STAMP benchmarks*, which were designed to be more representative of real-world behavior and more inclusive of diverse execution scenarios than microbenchmarks [36], and have continued to be used in more recent STM work (e.g., [55, 49]). We use a version of STAMP ported to Java by the Programming Languages Research Group at UC Irvine [51] and the Deuce authors [84]. We omit a few ported STAMP benchmarks because they run incorrectly, even when running single-threaded without STM on a commercial JVM. Six benchmarks run correctly, including two with both low- and high-contention workloads, for a total of eight benchmarks. Our experiments run the large workload size for all benchmarks except three. We run kmeans with twice the workload size of the standard large size, since otherwise load balancing issues thwart scaling significantly. labyrinth3d and ssca2 run out of memory on our implementation for the large workload size, so we choose parameters so the workload size is between the medium and large workloads. Although STAMP was originally written with hand-instrumented transactional loads and stores, our experiments do not use this information and instead instrument all transactional and non-transactional accesses, except for statically redundant barriers (Section 4.4) and a few known immutable types such as String.

To evaluate other implementations that target at lock-based programming model, we use the following benchmarks:

- *Benchmarked versions of large, real programs:* the DaCapo benchmarks, versions 2006-10-MR2 and 9.12-bach (2009) [18], excluding single-threaded programs and programs that Jikes RVM cannot execute, distinguished with suffixes 6 and 9.

- *Business logic benchmarks:* fixed-workload versions of SPECjbb2000 and SPECjbb2005.[8]

- *Medium-sized benchmarks that stress various multi-threaded execution scenarios:* the Java Grande benchmarks (excluding microbenchmarks) [139].

Platform. Experiments execute on an AMD Opteron 6272 platform and an Intel Xeon E5-4620 platform:

- Experiments for LarkTM-O, LarkTM-S, NOrec, and IntelSTM execute on an AMD Opteron 6272 system running RedHat Enterprise Linux 6.7, kernel 2.6.32. The system has eight 8-core processors (64 cores in total) that communicate via a NUMA interconnect. We have also repeated these performance experiments on a system with four Intel Xeon E5-4620 8-core processors (32 cores total). This platform supports NUMA, but we disable it for greater contrast with the AMD platform.

- Experiments for relaxed dependence tracking, RT-based recorder, RT-based STM, FastRCD-A, Valor-A, Snappy-P, Snappy-I execute on the Intel Xeon E5-4620 system.

[8] `http://www.spec.org/jbb200{0,5}`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

Chapter 3: Relaxed Dependence Tracking

This chapter investigates the potential for runtime support to hide latency introduced by dependence tracking, by tracking dependences in a *relaxed* way—meaning that not all dependences are tracked accurately. The key challenge in relaxing dependence tracking is to preserve both the program's semantics and the runtime support's guarantees. We present an approach called *relaxed dependence tracking* (RT) (Section 3.1) and demonstrate its potential by building two cases of RT-based runtime support (Section 3.2 and Section 5.1 [9]). Our evaluation shows that RT hides much of the latency incurred by dependence tracking, although RT-based runtime support incurs costs and complexity in order to handle relaxed dependence information (Section 3.3).

3.1 Tracking Cross-Thread Dependences under Relaxed Conditions

Section 2.1 described the approach for tracking dependences based on biased reader–writer locking. That approach tracks each cross-thread dependence soundly (i.e., does not miss any dependences), by waiting to acquire a read or write lock before proceeding with each access. In this chapter, we refer to that approach as *strict dependence tracking* (ST).

[9] We introduce RT-based STM after introducing the STM it builts upon in Chapter 4.

In contrast, this chapter introduces our novel approach called *relaxed dependence tracking* (RT), which relaxes the instrumentation–access atomicity guarantee provided by ST,[10] allowing threads to continue executing program code without acquiring a dependence tracking lock. The challenge in making RT work lies in preserving both program semantics and runtime-support-specific guarantees.

RT consists of two components: A *relaxed coordination protocol* (Section 3.1.1) and support for performing *relaxed accesses* that overlap with coordination (Section 3.1.2).

3.1.1 The Relaxed Coordination Protocol

In RT, a requesting thread does *not* wait for responses after sending requests. Thus, a requesting thread receives responses at some later point in its execution, and a requesting thread may have outstanding requests for multiple objects simultaneously. To support this functionality, the relaxed coordination protocol differs from the strict coordination protocol in the following ways:

- A responding thread can *respond* either implicitly or explicitly, depending on whether the *requesting* thread is blocking or actively executing program code.
- To support explicit responses, relaxed coordination extends strict coordination's request queue to a *request-and-response queue* that holds both requests and responses. At safe points, threads can receive not only requests, but also responses.

Figure 3.1 shows how the relaxed coordination protocol works. reqT sends an explicit request to respT and continues execution. When respT reaches a safe point, it responds to reqT either explicitly or implicitly. If reqT is blocked, respT responds implicitly, as shown

[10] Although our design of RT targets coordination latency introduced by biased reader–writer locking, it should be possible to adapt RT to other dependence-tracking mechanisms.

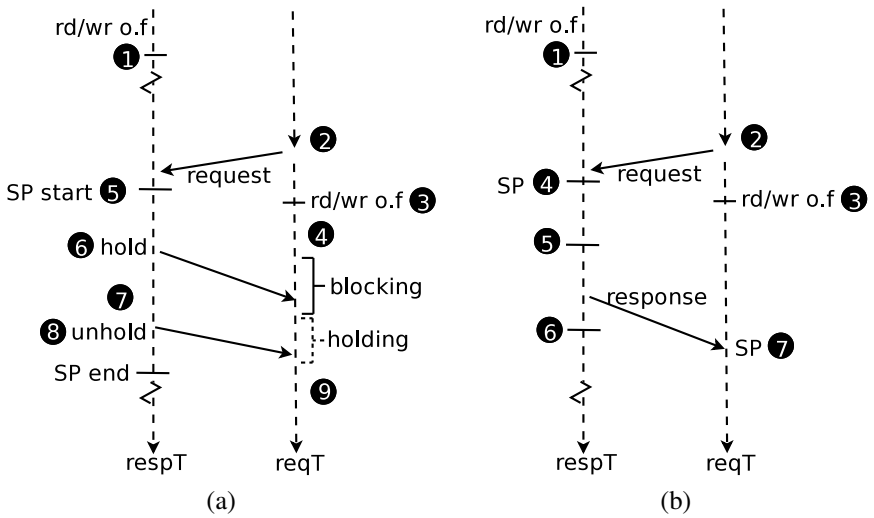


Figure 3.1: The *relaxed* coordination protocol. **(a) Implicit response:** (1) respT accessed o at some prior time. (2) reqT wants to access o. It changes o's lock to $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$. (3) reqT proceeds without waiting to receive respT's response. (4) respT reaches a safe point (SP) and (5) sees reqT in a blocking state, so respT puts reqT in a "blocked and held" state. (6) respT changes o's lock's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$. (7) respT removes the hold on reqT, then leaves the safe point. (8) reqT finishes blocking, then waits until all holds have been removed. **(b) Explicit response:** (1) respT accessed o at some prior time. (2) reqT wants to access o. It changes o's lock to $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$. (3) reqT proceeds without waiting to receive respT's response. (4) respT reaches a safe point (SP), (5) sends an explicit response, and (6) leaves the safe point. (7) reqT reaches a safe point and sees the response. reqT changes o's lock's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$.

in Figure 3.1, by first putting reqT into a "blocked and held" state (so that reqT does not leave the blocking state unless it is "unheld") and then changing the object's lock's state. Finally, the responding thread removes its hold on the requesting thread. Otherwise (reqT is not blocked), respT responds explicitly, as Figure 3.1 shows, by adding a response to reqT's queue. Once reqT reaches a safe point, it changes the object's lock's state.

Although Figure 3.1 shows a single requesting thread sending requests to respT, multiple requesting threads can send requests to respT before respT reaches a safe point. When respT reaches a safe point, it responds to each queued request in turn.

For a conflicting transition from $\mathsf{WrEx}_{\mathsf{respT}}$ or $\mathsf{RdEx}_{\mathsf{respT}}$ to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$, reqT receives just one response. In contrast, for a transition from RdSh to $\mathsf{WrEx}_{\mathsf{reqT}}$, reqT may need to wait for multiple responses. The protocol maintains a counter of unreceived responses for each object lock in this situation, which responding and requesting threads decrement as they respond implicitly and receive explicit responses, respectively.

Figures 3.2 and 3.3 show the pseudocode for RT's load and store instrumentation. RT uses the same fast path as ST (Section 2.1.1), except that it skips the original program access if it takes the slow path, delegating the access to the slow path instead. For both loads and stores, ST initiates coordination by changing the state of the object to $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ (line 14 in Figure 3.2) or $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{T}}$ (line 17 in Figure 3.3) and sending a request to the responding thread (line 15 in Figure 3.2 and line 18 in Figure 3.3). After sending the coordination request, T continues execution immediately, subject to constraints about accessing objects that are not yet locked in the needed state.

Since T does not wait for responses, it instead receives responses at safe points (not shown). A responding thread *responds* either implicitly or explicitly, depending on whether or not the requesting thread is at blocking safe point. Before T receives a response, the conflicting object o stays in the $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ or $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{T}}$ state, since both T and other threads might perform accesses to it. If the access is a store, and o is in $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{T}}$ state, RT upgrades the state to $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ (line 10 in Figure 3.2), in order to track relaxed stores, introduced shortly. If o is locked in $\mathsf{WrEx}^{\mathsf{Int}}_{*}$ but not $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ (i.e., other threads performing relaxed stores to o), the access needs to wait until the state has changed to a non-intermediate state. If the access is a load, as long as o is in $\mathsf{WrEx}^{\mathsf{Int}}_{*}$ or $\mathsf{RdEx}^{\mathsf{Int}}_{*}$, the access can avoid performing coordination.

We note that RT's relaxed coordination protocol differs from the strict coordination protocol for *explicit* requests only. In RT, when coordination uses an *implicit* request, it

```
1  if (o.state != WrEx_T) {
2    writeSlowPath(o, &o.f, newValue);
3  } else {
4    o.f = newValue; // original program store
5  }

6  writeSlowPath(o, addr, newValue) {
7    state = o.state;
8    // Handle non−conflicting state transitions :
9    if (state == ...) { ...; *addr = newValue; return; }
10   if (state == RdEx_T^Int) { /* upgrading trans. to WrEx_T^Int */ }
11   boolean relaxed = true;
12   while (state != WrEx_T^Int) {
13     if (state != WrEx_*^Int &&
14         CAS(&o.state, state, WrEx_T^Int)) {
15       relaxed = !sendCoordinationRequest(getOwner(state));
16       break;
17     }
18     state = o.state; // re−read state
19   }
20   if (relaxed) {
21     storeBufferSet(addr, newValue); // defer the store
22   } else {
23     o.state = WrEx_T;
24     *addr = newValue;
25   }
26 }
```

Figure 3.2: The fast and slow paths of RT's instrumentation at stores.

follows the same steps as strict coordination. Interestingly, the relaxed coordination protocol handles requests and responses in a largely symmetric way. Requests and responses each involve sending a message to another thread, either implicitly if the receiving thread is at a blocking safe point; or else explicitly via a queue that the receiving thread processes at its next safe point.

```
1  if (o.state != WrEx_T &&
2      o.state != RdEx_T &&
3      (o.state != RdSh_c || T.rdShCount < c))
4    ... = readSlowPath(o, &o.f); {
5  } else {
6    ... = o.f; // original program load
7  }

8  readSlowPath(o, addr) {
9    state = o.state;
10   // Handle non−conflicting state transitions :
11   if (state == ...) { ...; return *addr; }
12   if (state == WrEx_T^Int && storeBufferContains(addr))
13     return storeBufferGet(addr);
14   boolean relaxed = true;
15   while (state != WrEx_*^Int && state != RdEx_*^Int) {
16     // Coordination for conflicting transition :
17     if (CAS(&o.state, state, RdEx_T^Int)) {
18       relaxed = !sendCoordinateRequest(getOwner(state));
19       break;
20     }
21     state = o.state; // re-read state
22   }
23   value = *addr;
24   if (relaxed)
25     logLoadedValue(addr, value);
26   else
27     o.state = RdEx_T;
28   return value;
29 }
```

Figure 3.3: The fast and slow paths of RT's instrumentation at loads.

3.1.2 Handling Relaxed Accesses

A thread T performs *relaxed accesses*[11] to objects whose locks are not (yet) in the needed state. RT defers a *relaxed store* until it receives coordination response(s) for the object's lock. As we explain, relaxed stores still conform to the language memory model as long as they are not deferred past synchronization release operations. RT performs a

[11]This dissertation's *relaxed accesses* should not be confused with memory_order_relaxed operations on atomic variables in C/C++ [23].

relaxed load by loading from an object before receiving coordination response(s) for the object's lock. Relaxed loads do not affect program correctness, but they can affect runtime support's guarantees.

Relaxed stores. A thread T performs a relaxed store by *deferring* the store, buffering the location (address) and new value in T's *store buffer* (line 21 in Figure 3.2). The intuition behind deferring stores is that another thread may be simultaneously (racily) accessing the same location, so allowing the store to be performed could cause a cross-thread dependence to be missed. Once T gets exclusive ownership of the conflicting object o (by changing o's lock's state to $\mathsf{WrEx}_{\mathsf{T}}$), it performs all deferred stores to o using the store buffer.

For simplicity, our current design limits relaxed stores by T to objects locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ state. (We have found that supporting relaxed stores to other lock states provides little benefit.)

Deferring program stores changes program behavior since other threads can read out-of-date values from the affected memory locations. However, language memory models, including for Java and C++, allow substantial reordering of operations, except across synchronization operations [4, 23, 100, 6], thus permitting significant deferring of stores. To conform to the memory model and preserve program semantics, the key constraint is that stores *cannot* be deferred past program synchronization *release* operations (e.g., lock release, thread fork, and Java volatile or C++ atomic writes).

On the other hand, why would it be *incorrect* to allow stores to be deferred *past* synchronization release operations? Figure 3.4 shows an example. While object o's lock is intially in $\mathsf{WrEx}_{\mathsf{T1}}$, T1 sets o.f's value to v1. Next, T2 attempts to store v2 to o.f. T2 changes o's lock to $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T2}}$, sends an coordination request to T1, and defers the store in its store

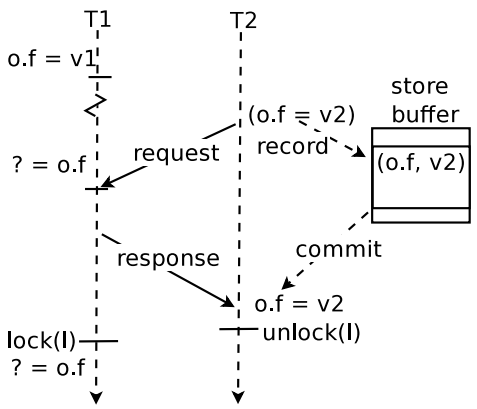


Figure 3.4: Relaxed stores cannot be deferred past synchronization points.

buffer, avoiding conflicting with any accesses by T1, which still "owns" o. T2 must commit its buffered store before executing unlock(l). Otherwise, T1's load after its lock(l) operation might read the old value v1, violating semantics, since communicating synchronization guarantees visibility [23, 100].

Relaxed loads. At a relaxed load by T to an object o, T first checks whether the same location has already been buffered in T's store buffer (line 12 in Figure 3.3). (T only needs to check its store buffer if o's lock is in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ state.) If so, T uses the store buffer's value (line 13 in Figure 3.3) instead of loading from memory. Otherwise, T performs the load directly from memory (line 23 in Figure 3.3). A relaxed load thus does *not* affect program semantics: the execution still conforms to the memory model (since performing the load would be permitted in the original program). However, a relaxed load could certainly impact the correctness of runtime support that needs to detect or control cross-thread dependences. In particular, another thread might be simultaneously (racily) writing to the same

memory location, compromising the ability of runtime support to capture the write–read or read–write dependence.

RT thus handles each relaxed load by *logging the loaded value* in a *runtime-support-specific way* (line 25 in Figure 3.3). The intuition is that logging the value enables runtime support to handle all values resulting from potentially untracked cross-thread dependences. For example, our RT-based dependence recorder logs the value in order to assist replay (Section 3.2), and our RT-based STM logs the value in order to validate it later (Section 5.1).

3.1.3 Optimizations at Synchronization Release Operations

As presented so far, a thread must wait at each program synchronization release operation for every outstanding deferred relaxed store (i.e., every entry in its store buffer). This restriction limits RT's ability to overlap coordination with program execution; we have found that threads routinely end a critical section (by releasing a lock) shortly after performing a store to a shared variable. Here we present two optimizations for avoiding waiting at release operations. As our evaluation shows, these optimizations have performance benefits but also drawbacks that lead to mixed performance relative to the base RT design described so far.

Defer release operations. Instead of waiting at a release operation for outstanding deferred stores, a thread can *defer the release operation*. Our design currently supports deferring *lock* release operations. To defer a lock release, a thread continues to hold the lock past the release operation; the thread records the lock into a per-thread *lock buffer*. It releases the lock only when all responses have been received for deferred relaxed stores that executed before the lock release (according to bookkeeping).

This behavior naturally preserves program semantics because a thread continues to hold a lock while it waits for responses for relaxed stores, effectively expanding the lock's critical section—making other threads wait and thus increasing lock contention. Effectively enlarging critical sections can serendipitously avoid some erroneous behaviors, which may be desirable or undesirable, depending on the goals and setting.

Avoid stalling at release. An alternate approach is to permit a thread T to continue execution at a release operation—as long as no other thread may access the object(s) that are the targets of deferred stores. A straightforward way to provide this restriction is to disallow all accesses by other threads to objects locked in $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Int}}$ state. (Note that, in contrast, the base RT design allows loads, but not stores, to an object in any intermediate state.)

This optimization allows threads to continue without waiting at release operations, but it incurs other costs because a thread T2 must wait to access an object locked in $\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Int}}$ state.

3.2 Recording Dependences

This section shows how to use RT to optimize *multi-threaded record & replay*, which enables deterministically replaying a recorded execution. Record & replay enables both offline debugging [89, 148] and system features such as replication-based fault tolerance [30, 90]. Recording dependences efficiently is the key challenge of multi-threaded record & replay. However, recording dependences is expensive due to the high cost of tracking dependences between all potentially shared memory accesses [89, 88].

3.2.1 ST-Based Dependence Recorder

Prior work builds a dependence recorder, which we call the *ST-based recorder*, on top of biased reader–writer locks that use strict dependence tracking [28]. It records all happens-before edges [87] at transitions between WrEx, RdEx, and RdSh states. It records each happens-before edge by recording its *source* and *sink*, each of which is recorded as a *dynamic program location* (DPL), which consists of a static program location (e.g., method and line number) and a per-thread counter incremented at every method entry, method exit, and loop back edge. Another execution can replay these happens-before edges deterministically by making the sink waits for its corresponding source to be reached.

3.2.2 RT-Based Dependence Recorder

Our *RT-based dependence recorder* extends the ST-based recorder by using relaxed, instead of strict, dependence tracking. The RT-based recorder allows relaxed loads and stores to objects that are not yet "owned" by the current thread. Given this behavior, how is it possible to record dependences accurately and thus guarantee deterministic replay?

We refer back to Figure 3.1 on page 43 for examples of happens-before edges recorded by the RT-based recorder. For an implicit response, at point #6, respT records the source of the edge. At point #8, reqT records the edge's sink. For an explicit response, respT records the source at point #5, and reqT records the sink at point #7. Note that if the replayed execution replays these same edges, it will not necessarily reproduce the same behavior because the relaxed accesses at point #3 (and other relaxed accesses potentially overlapping with coordination) are not ordered by the edge.

The key to addressing this problem is to record enough information about loads and stores that are *not* well-ordered by happens-before edges, such that they can be replayed faithfully.

Handling stores. To handle deferred stores to objects locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ state, the RT-based recorder uses the following strategy: reqT records an event for each deferred store, to indicate that the store should also be deferred *during replay*. When stores are performed from the store buffer at a safe point, reqT records an event indicating that deferred stores should be performed at that safe point. By referring to indices of entries in the store buffer, the recorded event unambiguously indicates *which* stores should be performed from the store buffer at each safe point during replay.

Handling loads. When a requesting thread reqT loads a value from an object whose lock is in an intermediate state, the responding thread may be simultaneously writing the object. Thus, it does *not* seem possible to record a happens-before edge that will yield the same value for the load. Instead, reqT *records the value* returned by the load (at point #3 in both Figure 3.1 and Figure 3.1). A replayed execution can reuse this value to ensure determinism.

During the recorded execution, subsequent loads to the same memory location record the (possibly updated) loaded value. Whenever reqT handles a load by getting the value from its store buffer (Section 3.1.2), it still records the value in its log, so a replayed execution can load the correct value without needing to know which loads should read from the store buffer.

Value determinism. Our RT-based recorder provides *value determinism*, i.e., each load reads the same value during replay as during record. If a load was relaxed during record, it will load from the log during replay. Notably, this value will match the value of the variable in memory (i.e., the value that *would* normally have been loaded from memory) unless there is a data race. Other efficient record & replay techniques, such as DoublePlay [148] and Respec [90], provide *output determinism*, which is weaker than value determinism but is still useful for both offline replay (e.g., replaying buggy executions) and online replay (e.g., replicating a multi-threaded process).

3.3 Evaluation

This section first evaluates the performance and runtime characteristics of relaxed dependence tracking (RT) without runtime support, compared with strict dependence tracking (ST). It then evaluates a case study of runtime support built upon RT, compared with its ST-based approach.

3.3.1 Evaluating Relaxed Dependence Tracking

This section evaluates RT and compares it with ST, by executing instrumentation that tracks dependences but provides no runtime support on top of it.

Measuring the problem. We first measure the cost of ST, as well as the maximum benefit that can be obtained from optimizations. Figure 3.5 shows runtime overhead of three configurations over an unmodified JVM. Each configuration instruments accesses to track dependences using biased reader–writer locks. The first configuration, *ST*, uses strict tracking (as described in Section 2.1.1.1) and adds 139% overhead on average. This overhead varies considerably across the evaluated programs; the overhead for each program is closely

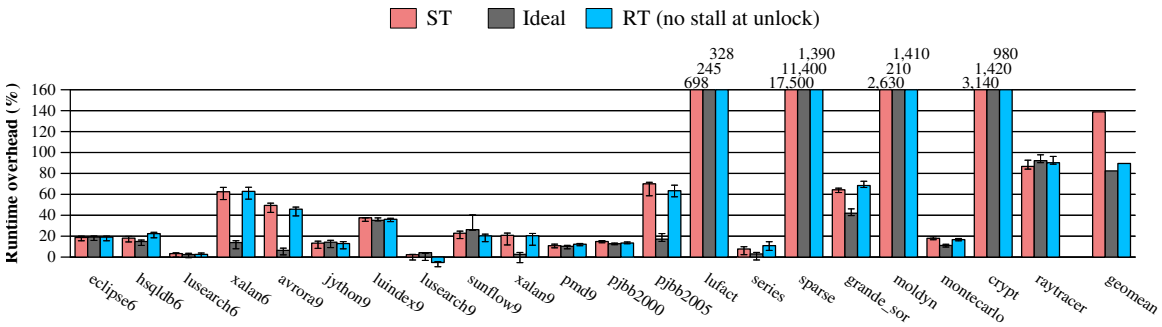


Figure 3.5: Runtime overhead added to an unmodified JVM by capturing dependences using (1) ST, compared with (2) an ideal, unsound configuration that eliminates coordination latency and (3) RT.

linked to the fraction of accesses that trigger coordination using explicit requests (as shown later in this section), which is the main cost of tracking dependences.

Ideal is an *unsound* configuration that eliminates most of the cost of strict coordination. In this configuration, after a thread sends an explicit request, it continues execution without waiting for any response. Responding threads in turn ignore requests. This configuration attempts to estimate an upper bound on the performance that RT might be able to provide. On average, *Ideal* adds 82% overhead—a little more than half of the overhead added by the sound configuration. The remaining costs are due to fast-path instrumentation at every access, as well as other transitions, including conflicting transitions that trigger coordination using implicit requests. In addition, although requesting threads do not wait for responses and responding threads ignore requests, *Ideal* incurs remote cache misses by sending explicit requests.

RT's effectiveness at hiding coordination costs. Figure 3.5's last configuration, *RT (no stall at unlock)*, uses relaxed dependence tracking with the second optimization from Section 3.1.3. Its overhead over baseline execution is 90%, a significant reduction (49% relative to baseline execution time) from ST's 139% overhead. Furthermore, RT achieves much of the maximum possible benefit, approaching *Ideal*'s 82% overhead.

We note that for sparse, RT significantly outperforms the *Ideal* configuration. This counterintuitive result is due to the fact that the *Ideal* configuration estimates the cost of coordination without latency, but it does not account for potential other improvements that RT might provide. As we show later in this section, RT can reduce the number of conflicting transitions (compared with ST); RT reduces sparse's conflicting transitions substantially, leading to significantly lower overhead than for *Ideal*.

Figure 3.6 is a *speedup* graph (higher is better) that shows the same configurations as Figure 3.5 plus two additional RT configurations. The RT configurations, which are all sound, are as follows:

RT (stall at unlock): The default design from Section 3.1. Threads wait at synchronization release operations for all outstanding relaxed stores.

RT (defer unlock): The first optimization described in Section 3.1.3. At a lock release, a thread defers the lock release if there are outstanding relaxed stores.

RT (no stall at unlock): The second optimization described in Section 3.1.3 (and also shown in Figure 3.5). At a lock release, a thread continues execution even if there are outstanding relaxed stores. However, no thread except T can read from an object locked in $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{T}}$ state.

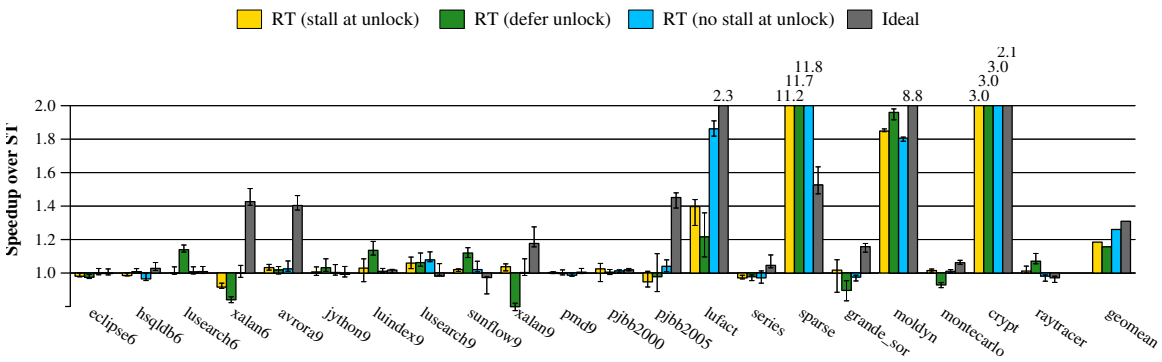


Figure 3.6: Speedup of RT relative to ST. *Ideal* is an unsound configuration that provides an upper bound on RT's performance.

The three RT configurations each provide an average speedup of 1.16–1.26X over ST. These speedups are not far from the average speedups achieved by *Ideal* (1.31X), suggesting that RT is getting most of the maximum possible benefit from hiding the cost of coordination due to explicit requests.

While the RT configurations outperform *ST* and get close to *Ideal*'s performance on average, RT does not help much with the gap between *ST* and *Ideal* in several cases (hsqldb6, xalan6, avrora9, xalan9, grand_sor, and montecarlo). As we show later, RT changes the balance of explicit versus implicit coordination requests (relative to ST), by causing threads to spend more time executing code instead of blocking at safe points. This change cancels out RT's potential performance benefits in several cases, and it represents a challenge for future work. Another significant source of RT overhead is bookkeeping costs: its queue representation leads to more costs than for ST, and keeping track of relaxed events and accesses requires performing additional work.

The *RT (defer unlock)* configuration does not improve performance on average (nor significantly for any individual program) compared with the *RT (stall at unlock)*. Although

deferring lock release operations has the potential to hide coordination latency, it incurs two additional costs. First, deferring releases incurs additional bookkeeping costs. Second, deferring releases often changes the balance between explicit and implicit requests triggered for coordination, since threads are more likely to be executing code rather than blocked at release operation waiting for coordination responses. These factors are enough to outweigh any potential benefit provided by deferring unlocks.

Similarly, the *RT (no stall at unlock)* configuration helps hide latency, but it introduces another source of latency: a thread (except for T) must wait to read an object locked in $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Int}}$ state. On average, these factors cancel each other, so *RT (no stall at unlock)* provides almost no average benefit over *RT (stall at unlock)*.

Runtime characteristics. Next we focus on understanding factors contributing to the performance difference between RT and ST. Table 3.1 reports runtime statistics for tracking dependences with RT and ST. The table uses the RT configuration *RT (no stall at unlock)*.

For each type of coordination, *Lock state transitions* counts how many accesses execute instrumentation that requires either no lock state change (*Same state*) or a *Conflicting* transition that triggers coordination. An interesting point is that RT sometimes reduces *how many* conflicting transitions occur, relative to ST. This phenomenon occurs because of cases in which an object is heavily contended, and two or more threads repeatedly transfer its ownership in quick succession. When using ST, a thread must wait for coordination at each access, enabling another thread to make progress and trigger coordination for the next access to the object, leading to many conflicting transitions. In contrast, when using RT—particularly when executing past release operations as permitted by the *RT no stall at unlock* configuration—a thread is more likely to perform consecutive *relaxed* accesses to a

	Strict dependence tracking				Relaxed dependence tracking							
	Lock state transitions		Coord. requests		Lock state transitions		Coord. requests		Coord. responses		Relaxed accesses	
	Same state	Conflicting	Explicit	Implicit	Same state	Conflicting	Explicit	Implicit	Explicit	Implicit	Read	Write
eclipse6	1.2×10^{10}	1.4×10^{5} (0.0011%)	1.6×10^{4}	2.9×10^{5}	1.2×10^{10}	1.4×10^{5} (0.0011%)	1.1×10^{4}	2.6×10^{5}	9.6×10^{3}	1.4×10^{3}	1.4×10^{4}	4.8×10^{3}
hsqldb6	6.2×10^{8}	9.0×10^{5} (0.14%)	3.5×10^{4}	3.8×10^{6}	6.2×10^{8}	9.0×10^{5} (0.14%)	3.4×10^{4}	4.4×10^{6}	3.1×10^{4}	3.5×10^{3}	4.7×10^{4}	3.8×10^{4}
lusearch6	2.4×10^{9}	4.4×10^{3} (0.00018%)	2.3×10^{3}	4.5×10^{3}	2.4×10^{9}	4.4×10^{3} (0.00018%)	2.3×10^{3}	4.6×10^{3}	8.8×10^{2}	1.4×10^{3}	2.2×10^{3}	2.1×10^{3}
xalan6	1.1×10^{10}	1.9×10^{7} (0.17%)	1.3×10^{7}	5.9×10^{6}	1.1×10^{10}	1.9×10^{7} (0.17%)	1.4×10^{7}	5.2×10^{6}	1.3×10^{7}	6.3×10^{5}	1.6×10^{7}	1.8×10^{7}
avrora9	6.1×10^{9}	5.9×10^{6} (0.097%)	4.1×10^{6}	1.8×10^{7}	6.1×10^{9}	5.7×10^{6} (0.093%)	2.8×10^{6}	1.4×10^{7}	2.2×10^{6}	5.5×10^{5}	2.1×10^{6}	1.9×10^{6}
jython9	5.1×10^{9}	6.6×10^{1} (0.0000013%)	1.8×10^{1}	1.5×10^{0}	5.1×10^{9}	6.2×10^{1} (0.0000012%)	1.5×10^{1}	4.5×10^{0}	1.3×10^{1}	2.0×10^{0}	2.3×10^{1}	0
luindex9	3.5×10^{8}	3.7×10^{2} (0.00011%)	1.5×10^{1}	3.3×10^{2}	3.5×10^{8}	3.7×10^{2} (0.00011%)	1.2×10^{1}	3.3×10^{2}	9.5×10^{0}	3.0×10^{0}	1.9×10^{1}	2.5×10^{0}
lusearch9	2.4×10^{9}	2.9×10^{3} (0.00012%)	4.6×10^{3}	4.4×10^{3}	2.4×10^{9}	2.9×10^{3} (0.00012%)	5.0×10^{3}	3.3×10^{3}	1.3×10^{3}	3.7×10^{3}	6.4×10^{3}	4.1×10^{2}
sunflow9	1.7×10^{10}	1.4×10^{4} (0.000078%)	1.5×10^{4}	7.6×10^{3}	1.7×10^{10}	9.3×10^{3} (0.000054%)	9.6×10^{3}	8.7×10^{3}	3.3×10^{3}	6.3×10^{3}	2.4×10^{5}	8.4×10^{3}
xalan9	1.0×10^{10}	1.8×10^{7} (0.18%)	9.7×10^{6}	8.7×10^{6}	1.0×10^{10}	2.0×10^{7} (0.20%)	1.3×10^{7}	7.1×10^{6}	1.3×10^{7}	6.4×10^{5}	2.0×10^{7}	2.1×10^{7}
pmd9	5.7×10^{8}	4.4×10^{4} (0.0077%)	3.1×10^{4}	5.3×10^{4}	5.7×10^{8}	4.3×10^{4} (0.0075%)	2.7×10^{4}	4.9×10^{4}	2.0×10^{4}	6.9×10^{3}	2.5×10^{4}	3.4×10^{4}
pjbb2000	1.7×10^{9}	9.5×10^{5} (0.055%)	6.2×10^{4}	9.0×10^{5}	1.7×10^{9}	9.5×10^{5} (0.055%)	6.1×10^{4}	9.0×10^{5}	5.7×10^{4}	3.5×10^{3}	2.3×10^{5}	9.7×10^{4}
pjbb2005	6.6×10^{9}	4.6×10^{7} (0.69%)	3.2×10^{7}	5.7×10^{7}	6.5×10^{9}	4.1×10^{7} (0.61%)	2.5×10^{7}	6.2×10^{7}	1.9×10^{7}	5.5×10^{6}	1.2×10^{7}	1.6×10^{7}
lufact	8.0×10^{9}	6.0×10^{5} (0.0075%)	5.3×10^{6}	1.4×10^{6}	8.4×10^{9}	5.2×10^{5} (0.0061%)	8.5×10^{5}	8.4×10^{5}	3.3×10^{5}	5.3×10^{5}	1.2×10^{7}	3.1×10^{5}
series	4.0×10^{6}	2.0×10^{6} (33%)	2.0×10^{6}	3.0×10^{4}	4.0×10^{6}	2.0×10^{6} (33%)	1.4×10^{6}	6.2×10^{5}	1.2×10^{6}	1.4×10^{5}	1.2×10^{2}	1.4×10^{6}
sparse	6.7×10^{9}	2.4×10^{8} (3.4%)	3.8×10^{8}	8.3×10^{7}	6.0×10^{9}	4.5×10^{7} (0.74%)	3.1×10^{7}	1.8×10^{7}	2.8×10^{7}	3.3×10^{6}	5.0×10^{8}	4.9×10^{8}
grande_sor	3.7×10^{9}	3.9×10^{4} (0.0011%)	4.2×10^{5}	1.2×10^{5}	3.6×10^{9}	3.9×10^{4} (0.0011%)	3.7×10^{5}	1.3×10^{5}	4.2×10^{4}	3.3×10^{5}	8.8×10^{4}	2.8×10^{4}
moldyn	4.0×10^{10}	9.7×10^{7} (0.25%)	1.7×10^{8}	8.0×10^{7}	3.3×10^{10}	8.5×10^{7} (0.26%)	7.8×10^{7}	5.3×10^{7}	4.9×10^{7}	2.9×10^{7}	6.6×10^{7}	5.3×10^{7}
montecarlo	2.4×10^{9}	5.2×10^{5} (0.022%)	3.2×10^{5}	2.0×10^{5}	2.4×10^{9}	4.3×10^{5} (0.018%)	2.7×10^{5}	1.5×10^{5}	2.5×10^{5}	2.2×10^{4}	2.2×10^{5}	3.1×10^{5}
crypt	5.2×10^{8}	3.9×10^{7} (7.0%)	3.9×10^{7}	3.2×10^{5}	5.2×10^{8}	7.4×10^{6} (1.4%)	4.8×10^{6}	2.6×10^{6}	4.8×10^{6}	1.1×10^{4}	2.2×10^{3}	3.7×10^{7}
raytracer	3.3×10^{10}	1.1×10^{4} (0.000032%)	7.1×10^{3}	5.2×10^{3}	3.3×10^{10}	1.1×10^{4} (0.000032%)	5.6×10^{3}	6.4×10^{3}	4.4×10^{3}	1.2×10^{3}	1.6×10^{5}	1.6×10^{3}

Table 3.1: Runtime characteristics of strict and relaxed dependence tracking.

contended object, leading to fewer conflicting transitions, compared with ST. This effect is directly responsible for RT outperforming the *Ideal* configuration for sparse.

The *Coord. requests* columns count explicit and implicit requests, which can sum to more than *Conflicting* transitions because RdSh-to-WrEx transitions involve multiple requests. Programs with more explicit requests generally have higher coordination overhead and can benefit more from RT.

The *Coord. responses* columns tally RT responses. Each sum equals the number of explicit requests, since there is one response for every explicit request. Since explicit responses do not incur latency, the ratio of explicit to implicit responses does not affect performance significantly.

The last two columns count relaxed accesses. While some of these accesses immediately follow coordination requests, others are repeat accesses to the same memory location or loads from objects for which some *other* thread has initiated coordination. Due to these

cases, relaxed accesses can outnumber conflicting transitions. On the other hand, conflicting transitions can outnumber relaxed accesses, since an implicit request does not lead to a relaxed access.

3.3.2 Performance of Runtime Support

This section evaluates whether RT can benefit runtime support that detects cross-thread dependences (dependence recorder) and controls cross-thread dependences (STM).

Dependence recorder. Figure 3.7 shows the performance of the ST- and RT-based recorders. Not surprisingly, the performance story for the recorders is similar to the story for tracking dependences alone. On average, the RT recorder is 1.23X faster than the ST-based recorder, and four benchmarks show large improvements.

We note that although RT can hide coordination latency, the RT-based recorder still needs to record each happens-before edge. Some relaxed loads can avoid conflicting transitions and coordination entirely, but the recorder must log each of the loaded values. The RT-based recorder thus often logs *more* than the ST-based recorder. Notably, the RT-based recorder's log size is about 2X the ST-based recorder's for lusearch6, xalan6, and xalan9; and about 6X for grande_sor. For all other programs, the RT-based recorder logs less than 50% more than the ST-based recorder.

In summary, RT reduces the cost of tracking dependences significantly, particularly for programs with high coordination costs, recovering much of the maximum possible performance achievable by eliminating coordination latency entirely. RT's benefit is limited by correctness constraints (e.g., limitations on deferring stores past synchronization) and indirect effects (e.g., increases to the explicit-to-implicit request ratio when using RT compared with ST). Although the RT-based recorder suffers drawbacks (increased log size),

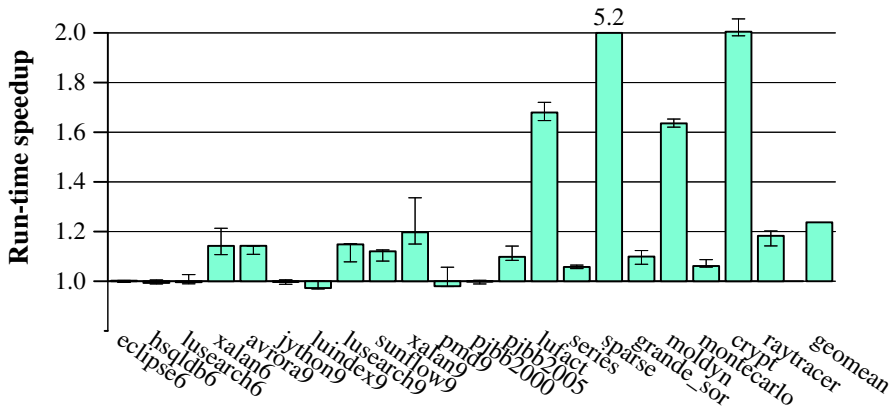


Figure 3.7: Runtime speedup of the RT-based dependence recorder over the ST-based dependence recorder.

these results demonstrate the potential of RT to improve the performance of dependence-tracking-based runtime support.

3.4 Conclusion and Interpretation

Many runtime support for concurrent programming build on top of memory dependence tracking. Existing work that track memory dependences synchronously can add high overhead because of coordination. In this chapter, we introduce RT, which is a novel approach for tracking cross-thread memory dependences. RT hides synchronization latency by tracking dependences in a relaxed way—meaning that not all dependences are tracked accurately. Our key insight is to relax the instrumentation-access atomicity provided by normal tracking, and to allow threads to continue executing program code without acquiring a dependence tracking lock.

The challenge for this work is to preserve both program semantics and runtime-support-specific guarantees after making the relaxation. We use log buffering to defer relaxed stores

so their effect is invisible to other threads until the coordination response(s) for the object's lock has been received. We handle relaxed loads by logging and processing the loaded value in a runtime-support-specific way. We use *lazy lock release* techniques to extend critical sections to allow more code being executed while a thread is waiting for responses because Java memory model requires all states to become visible after a lock release.

Our performance results show that the relaxed approach has lower overhead than the strict tracking approach, especially for high-contention cases. Drawbacks include increased complexity and space overhead in order to handle extra relaxed dependence information. We show RT is directly useful for improving record & replay's performance, and it is applicable to other runtime support, such as transactional memory introduced in Chapter 4. From a research prospective, RT demonstrates that it is possible to build runtime support more efficietly with relaxed dependence tracking while still preserving correctness. This work therefore shows the potential to address a key cost of building efficient and scalable runtime support for parallelism.

Chapter 4: LarkTM

Transactional memory provides a new concurrency control option that can avoid the perils of locks and eases concurrent programming significantly. By avoiding deadlocks and allowing fine-grained concurrency, TM supports the programmer to write scalable applications safely.

Despite being promising, bringing TM into the mainstream faces many challenges. STM suffers from high overhead and weak semantics in practice. This chapter presents a novel STM system, called LarkTM, which incurs low overhead while at the same time supports strong semantics and strong progress guarantees. Section 4.1 introduces the design of LarkTM-O. LarkTM-O adds low overhead and scales well with low contention programs by reducing one of the major costs from detecting transactional conflicts (i.e., write–read, write–write, and read–write data dependences involving two transactions). Section 4.2 introduces the design of LarkTM-S, which extends from LarkTM-O and uses a cost-benefit model to adaptively switch concurrency control on *contended objects* from using optimistic concurrency control to pessimistic concurrency control. LarkTM-S adds little overhead on top of LarkTM-O but is able to make concurrent programs scale with high-contention workloads. Section 4.4 gives the implementation details not covered in Section 2.4. Section 4.5 evaluates LarkTM-O and LarkTM-S against existing high-performance STMs.

4.1 LarkTM-O: Low-Overhead Software Transactional Memory

This section describes a novel STM called LarkTM that uses eager mechanisms—eager versioning and eager conflict detection and resolution—and provides strong atomicity. LarkTM extends lightweight reader–writer locks to detect transactional conflicts. The locks add minimal overhead to non-conflicting accesses and only cause synchronization overhead at conflicting accesses, incurring very low overhead for programs with relatively good thread localities. Further, LarkTM enables eager conflict detection and resolution that support to abort either conflicting thread, providing strong progress guarantees.

LarkTM's approach works well under low contention, but turns out to be too aggressive under higher contention. Section 4.2 will describe an adaptive version of LarkTM that uses pessimistic locking for high-contention objects.

4.1.1 Design Overview

LarkTM provides concurrency control by acquiring the biased reader–writer lock introduced in Section 2.1.1 before each transactional and non-transactional memory access. Figure 4.1 shows at a high level how the instrumentation for acquiring a lock proceeds. If the current thread already holds the needed lock, there is no *transactional conflict* (a conflict between a transaction and either another transaction or a non-transactional access), and the access can proceed without performing synchronization. A thread does not need to release the biased lock even when a transaction ends. Instead, a biased lock is released only when there is another transactional access or non-transactional access that needs to acquire the same lock to access the conflicting object.

If the current thread does not already hold the needed biased read or write lock, it must acquire the lock. In some cases, acquiring the lock simply involves "upgrading" it, e.g.,

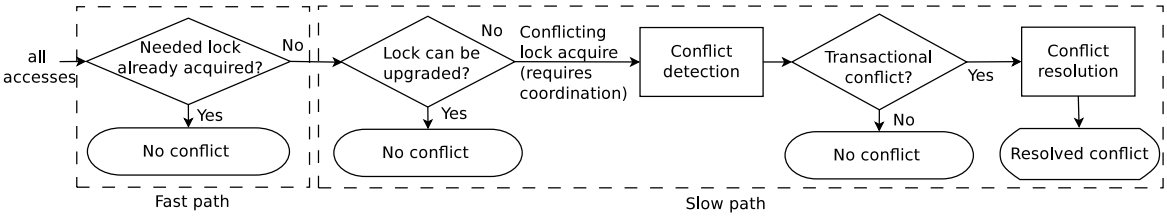


Figure 4.1: Overview of LarkTM's concurrency control. The figure shows the process of acquiring a lock for every transactional and non-transactional access.

from read-exclusive to read-shared access. However, in other cases, the new lock status will *conflict* with the old lock status (e.g., a read by one thread to an object held by another thread for write-exclusive access). These cases require that the new thread *coordinate* with the old thread(s) before acquiring the lock. Otherwise, the old thread(s) might continue to access the object—after all, they will not use any synchronization as long as they see the old lock status. This change of a lock's state is called a *conflicting lock transition* or simply a *conflicting transition*.

A conflicting lock transition is a necessary condition for a transactional conflict, and so LarkTM checks for transactional conflicts (and performs conflict resolution if it detects a transactional conflict). Since a conflicting transition is an insufficient condition for a transactional conflict, LarkTM maintains *read/write sets* (i.e., the last transaction(s) to access each object) to make conflict detection more precise.

4.1.2 Conflict Detection

Section 2.1.1.1 presents how biased reader–writer locking tracks cross-thread dependences and handles conflict transitions with coordination. This part describes how LarkTM extends from detecting dependences to detecting transactional conflicts. In order to simplify the description, we first introduce *active and passive threads*.

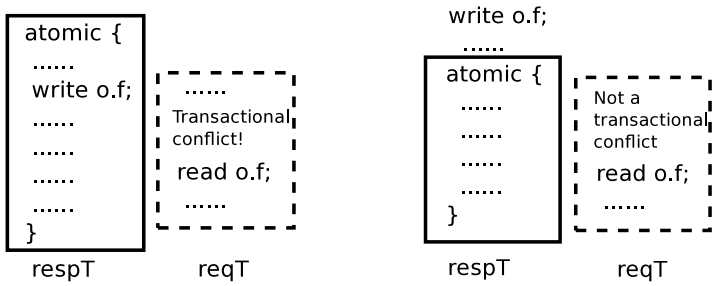


Figure 4.2: A conflicting transition is a necessary but insufficient condition for a transactional conflict. Solid boxes are transactions; dashed boxes could be either transactional or non-transactional.

Active and passive threads. Unlike many existing STMs, LarkTM allows either thread involved in a conflicting transition to perform transactional conflict detection. We refer to the thread that performs transactional conflict detection and resolution as the *active* thread. Either the requesting or responding thread is the active thread, depending on the protocol. The other thread is the *passive* thread.

	Active thread	Passive thread
Implicit protocol	Requesting thread	Responding thread
Explicit protocol	Responding thread	Requesting thread

These assignments make sense as follows. During the implicit protocol, the responding thread is blocked, so the requesting thread must do all the work. In the explicit protocol, the responding thread performs concurrency control because the requesting thread is stopped while it responds.

During the coordination protocol, the active thread detects transactional conflicts while the passive thread is stopped. Figure 4.2 shows how an object's lock's conflicting transition (a) may or (b) may not indicate a transactional conflict, since the responding thread's current transaction (if any) may or may not have accessed the object.

To detect whether the responding thread has accessed the object, LarkTM maintains read/write sets. For an object locked in the $\mathsf{WrEx_T}$ or $\mathsf{RdEx_T}$ state, it maintains the last transaction of T to access the object. Our implementation maintains this information in object headers. For an object locked in the RdSh state, LarkTM maintains each thread's last transaction to read the object since being locked in the RdSh state. Our implementation maintains this information using per-thread sets of objects read by the current transaction.

The active thread detects transactional conflicts by using the read/write sets to identify the last transaction (if any) of respT to access the object whose lock is changing states. If this transaction is the same as respT's current transaction (if any), the active thread has identified a transactional conflict, so it triggers conflict resolution.

An alternative is to let LarkTM simply assume a transactional conflict on every conflicting transition, avoiding the need for maintaining read/write sets. However, we have found that in practice, this approach leads to too many false conflicts, hurting scalability.

Detecting conflicts at WrEx→RdEx. It is challenging to detect conflicts precisely at a read by reqT to an object whose lock is in $\mathsf{WrEx_{respT}}$ state. Consider Figure 4.3(a). Object o's lock is initially in $\mathsf{WrEx_{respT}}$ state. respT's transaction reads but does not write o. Then reqT performs a conflicting access, changing o's lock's state to $\mathsf{RdEx_{reqT}}$. In theory, conflict detection need *not* report a transactional conflict. However, if reqT later writes to o, as in Figure 4.3(b), upgrading the lock's state to $\mathsf{WrEx_{reqT}}$, conflict detection *should* report a conflict with respT. It is hard to detect this conflict at reqT's write, since o's prior access information has been lost (replaced by reqT). The same challenge exists regardless of whether reqT executes its read and write in or out of transactions.

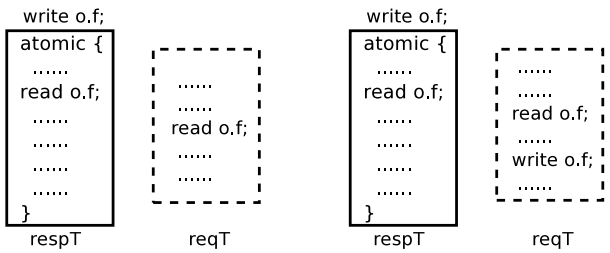


Figure 4.3: (a) Thread reqT's read triggers a state change from $\mathsf{WrEx}_{\mathsf{respT}}$ to $\mathsf{RdEx}_{\mathsf{reqT}}$, at which point LarkTM declares a transactional conflict even though respT's transaction has only read, not written, o. This imprecision is needed because otherwise (b) reqT might write o later, triggering a true transactional conflict that would be difficult to detect at that point.

One way to handle this case *precisely* is to transition a lock to RdSh in cases like reqT's read in Figures 4.3(a) and 4.3(b), when respT's transaction has read but not written the object. This precise policy triggers a $\mathsf{RdSh} \rightarrow \mathsf{WrEx}_{\mathsf{reqT}}$ transition at reqT's write in Figure 4.3(b), detecting the transactional conflict.

However, the precise policy can hurt performance by leading to more RdSh→WrEx transitions. LarkTM thus uses an *imprecise* policy: for a conflicting read (i.e., a read to an object locked in another thread's WrEx state), the active thread checks whether respT's transaction has performed writes *or reads*. Thus, in Figures 4.3(a) and 4.3(b), LarkTM detects a transactional conflict at reqT's conflicting read. We find that LarkTM's imprecise policy impacts transactional aborts insignificantly compared to the precise policy, except for the STAMP benchmark kmeans, for which the imprecise policy triggers 30% fewer aborts—but kmeans has a low abort rate to begin with, so its performance is unchanged. Overall, the precise policy hurts performance by leading to more RdSh→WrEx transitions.

We emphasize that LarkTM's imprecise policy for handling conflicting reads does *not in general* lead to concurrent reads generating false transactional conflicts. Rather, false

conflicts occur only in cases like Figure 4.3(a), where o's lock is in $\mathsf{WrEx}_{\mathsf{respT}}$ state because respT has previously written o, but respT's current transaction has only read, not written, o.

4.1.3 Resolving Transactional Conflicts

If an active thread detects a transactional conflict, it triggers conflict resolution, which resolves the conflict by aborting a transaction or retrying a non-transactional access. A key feature of LarkTM is that, by piggybacking on coordination, it can abort either conflicting thread, enabling flexible conflict resolution.

Contention management. When resolving a conflict, the active thread can abort either thread, providing flexibility for using various contention management policies [141]. LarkTM uses an *age-based* contention management policy [78] that chooses to abort whichever transaction or non-transactional access started more recently. This policy provides not only livelock freedom but also starvation freedom: each thread's transaction will eventually commit (a repeatedly aborting transaction will eventually be the oldest) [141]. We experimented with other contention management policies, including the Karma policy from prior work [127], but found that they had little affect on scalability, at least for benchmarks we evaluated.

Aborting a thread. The *aborting* thread abortingT chosen by contention management may be executing a transaction or a non-transactional access's lock acquire. "Aborting" a non-transactional access means retrying its preceding lock acquire.

To ensure that only one thread at a time tries to roll back abortingT's stores, the active thread first acquires a lock for abortingT. Note that another thread otherT can initiate implicit coordination with abortingT while abortingT's stores are being rolled back. If otherT triggers

coordination in order to access an object that is part of abortingT's speculative state, otherT will find the object locked in $\mathsf{WrEx}_{\mathsf{abortingT}}$ state, triggering conflict resolution, which will wait on abortingT's lock until rollback finishes.

In work tangentially related to piggybacking conflict resolution on coordination, Harris and Fraser present a technique that allows a thread to revoke a second thread's lock without blocking [74].

Handling the conflicting object. When conflict resolution finishes, the conflicting object's lock is still in the intermediate state $\mathsf{Int}_{\mathsf{reqT}}$. If abortingT is respT, then reqT changes the lock's state to $\mathsf{WrEx}_{\mathsf{reqT}}$ or $\mathsf{RdEx}_{\mathsf{reqT}}$. If abortingT is reqT, then the active thread *reverts* the lock's state back to its original state ($\mathsf{WrEx}_{\mathsf{respT}}$, $\mathsf{RdEx}_{\mathsf{respT}}$, or RdSh), after rolling back speculative stores. This policy makes sense because reqT is aborting, but respT will continue executing. (The lock cannot stay in the $\mathsf{Int}_{\mathsf{reqT}}$ state since that would block other threads from ever accessing it.)

Retrying transactions and non-transactional accesses. After the active thread rolls back the aborting thread's speculative stores, and the lock state change completes or reverts, both threads may continue. The aborting thread sees that it should abort, and it retries its current transaction or non-transactional access.

4.1.4 LarkTM's Instrumentation

The following pseudocode shows the instrumentation that LarkTM adds to every memory access to acquire a per-object reader–writer lock and perform other STM operations. At a program write:

```
1   if (o.state != WrEx_T) { // fast-path check
2     // Acquiring lock requires changing its state;
3     // conflicting acquire → conflict detection
4     slowPath(o);
5   }
6   // Update read/write set (if in a transaction):
7   o.lastAccessingTx = T.currentTx;
8   // Update undo log (if in a transaction):
9   T.undoLog.add(&o.f);
10  o.f = ...; // program write
```

At a program read:

```
11  if (o.state != WrEx_T && o.state != RdEx_T) { // fast-path
12    if (o.state != RdSh) {                      // check
13      // Acquiring lock requires changing its state;
14      // conflicting acquire → conflict detection
15      slowPath(o);
16    }
17    load_fence; // ensure RdSh visibility
18  }
19  // Update read/write set (if in a transaction):
20  if (o.state == RdSh)
21    T.sharedReads.add(o);
22  else
23    o.lastAccessingTx = T.currentTx;
24  ... = o.f; // program read
```

The fast-path check corresponds to the first three rows in Table 2.1. If the fast-path check fails, acquiring the lock requires a state change. If the state change is conflicting, it triggers the coordination protocol and transactional conflict detection. After line 5 (for writes) or 18 (for reads), the instrumentation has acquired the lock in a state sufficient for the pending access. For transactional accesses only, the instrumentation adds the object access to the transaction's read/write set. For an object locked in WrEx or RdEx, each object keeps track of its last accessing transaction; for an object locked in RdSh, each thread tracks the objects it has read (Section 4.1.2). Then, for transactional writes only, the instrumentation records the memory location's old value in an *undo log*. Finally, the access proceeds.

LarkTM naturally provides strong atomicity by acquiring its locks at non-transactional as well as transactional accesses. While one could implement weakly atomic LarkTM

by eliding non-transactional instrumentation, the semantics would be weaker than SLA (Section 2.2.2), e.g., the resulting STM would not be privatization or publication safe.

Redundant instrumentation. LarkTM can avoid statically *redundant* instrumentation to the same object in the same transaction, which can be identified by intraprocedural compile-time dataflow analysis [28]. Instrumentation at a memory access is redundant if it is definitely preceded by a memory access that is at least as "strong" (a write is stronger than a read). Outside of transactions, LarkTM can avoid instrumenting redundant lock acquires in regions bounded by safe points, since safe points interrupt atomicity [28].

4.2 LarkTM-S: Scalable Software Transactional Memory

As described so far, LarkTM minimizes overhead by making non-conflicting lock acquires as fast as possible. However, conflicting lock acquires—which can significantly outnumber actual transactional conflicts—require expensive coordination. To address this challenge, we introduce *LarkTM-S*, which targets better scalability. We call the "pure" configuration described so far *LarkTM-O* since it minimizes overhead.

A contended lock state. To support LarkTM-S, we add a new *contended* lock state to LarkTM's existing WrEx_T, RdEx_T, and RdSh states. Our current design uses IntelSTM's concurrency control [135] (Section 2.2.2) for the contended state. IntelSTM and LarkTM are fairly compatible because they both use eager concurrency control for writes. Following IntelSTM, LarkTM-S uses unbiased locks for writes to objects in the contended state, incurring an atomic operation for every non-transactional write and every transaction's first write to an object, but never requiring coordination. For reads to an object locked in the

contended state, LarkTM-S uses lazy validation of the object's *version*, which is updated each time an object's write lock is acquired.

Our current design supports changing an object's lock to the contended state at allocation time or as the result of a conflicting lock acquire. It is safe to change a lock to contended state in the middle of a transaction because coordination resolves any conflict, guaranteeing all transactions are consistent up to that point.

Profile-guided policy. LarkTM-S decides which objects' locks to change to the contended state based on profiling lock state changes. It uses two profile-based policies. The first policy is object based: if an object's lock triggers "enough" conflicting lock acquires, the policy puts the lock into the contended state. This policy counts each lock's conflicts at run time; if a count exceeds a threshold, the lock changes to contended state. (We would rather compute an object's ratio of conflicts to all accesses, but counting *all* accesses at run time would be expensive.)

The object-based policy works well except when many objects trigger few conflicts each. The second, type-based policy addresses this case by identifying object types that contribute to many conflicts. The type-based policy decides whether all objects of a given type (i.e., Java class) should have their locks put in the contended state at allocation time. For each type, the policy decides to put its locks into the contended state if, across all accesses to objects of the type, the ratio of conflicting to all accesses exceeds a threshold. Our implementation uses offline profiling; a production-quality implementation could make use of online profiling via dynamic recompilation. Grouping by type enables allocating objects locked in contended state, but the grouping may be too coarse grained, conflating distinct object behaviors.

Prior work has also adaptively used different kinds of locking for high-conflict objects, based on profiling [34, 147].

Semantics and progress. Since LarkTM-S validates reads lazily, it permits so-called *zombie* transactions [75]. Zombie transactions can throw runtime exceptions or get stuck in infinite loops that would be impossible in any unserializable execution. Each transaction must validate its reads before throwing any exception, as well as periodically in loops, to handle erroneous behavior that would be impossible in a serializable execution.

Since our design targets managed languages that provide memory and type safety, zombie transactions *cannot* cause memory corruption or other arbitrary behaviors [104, 54, 47]. A design for unmanaged languages (e.g., C/C++) would need to check for unserializable behavior more aggressively [47].

Like IntelSTM and other mixed-mode STMs, LarkTM-S can suffer livelock, since any transaction that fails read validation must abort (Section 2.2.2). Standard techniques such as *exponential backoff* [141, 78] help to alleviate this problem. We note that LarkTM-S can in fact *guarantee* livelock and starvation freedom by forcing a repeatedly aborting transaction to fall back to using entirely eager mechanisms (as though it were executed by LarkTM-O).

4.3 Comparison with NOrec and IntelSTM

To enhance our evaluation, we implement and compare against two STMs from prior work: NOrec [49] and IntelSTM (the strongly atomic version of McRT-STM) [124, 135] (Section 2.2.2). NOrec is generally considered to be a state-of-the-art STM (e.g., recent work compares quantitatively against NOrec [33, 77, 152]) that provides relatively low

	NOrec	IntelSTM	LarkTM-O	LarkTM-S
Write concurrency control	Lazy global seqlock	Eager per-object lock	Eager per-object biased reader–writer lock	IntelSTM–LarkTM-O hybrid
Read concurrency control	Lazy value validation	Lazy version validation	Eager per-object biased reader–writer lock	IntelSTM–LarkTM-O hybrid
Instrumented accesses	All accesses	Non-redundant accesses	Non-redundant accesses	Non-redundant accesses
Progress guarantee	Livelock free	None	Livelock and starvation free	Livelock and starvation free*
Semantics	SLA	Strong atomicity	Strong atomicity	Strong atomicity

Table 4.1: Comparison of the features and properties of NOrec [49], IntelSTM [135], LarkTM-O, and LarkTM-S. SLA is single global lock atomicity (Section 2.2.2). *LarkTM-S guarantees progress only if it forces a repeatedly aborting transaction to use fully eager concurrency control.

single-thread overhead and (for many workloads) good scalability. Although not considered to be one of the best-performing STMs, IntelSTM is perhaps the highest performance STM from prior work that supports strong atomicity.

Table 4.1 compares features and properties of our STMs and prior work's STMs. LarkTM uses biased reader–writer locks for concurrency control to achieve low overhead. NOrec and IntelSTM use lazy validation for reads in order to avoid the overhead of locking at reads, but as a result they incur other overheads such as logging reads (both), looking up reads in the write set (NOrec), and validating reads (IntelSTM).

IntelSTM, LarkTM-O, and LarkTM-S can avoid redundant concurrency control instrumentation (Section 4.1.4) because they use object-level locks and/or version validation. NOrec must instrument all reads fully since it validates reads using values; NOrec performs only logging (no concurrency control) at writes. None of the STMs can avoid logging at redundant writes because we have implemented an object-granularity dataflow analysis (Section 4.4).

NOrec provides livelock freedom (i.e., some thread's transaction eventually commits), and IntelSTM makes no progress guarantees. LarkTM-O provides starvation freedom (every transaction eventually commits) by resolving conflicts eagerly and supporting aborting

either transaction. LarkTM-S can provide starvation freedom if it uses (LarkTM-O's) fully eager concurrency control for a repeatedly aborting transaction.

NOrec provides weak atomicity (SLA; Section 2.2.2); a strongly atomic version would need to acquire a global lock at every non-transactional store. The other STMs provide strong atomicity by instrumenting each non-transactional access like a tiny transaction.

4.4 Implementation

Programming model. Our design assumes that the programmer only needs to add atomic blocks around code to make it execute as a transaction. Our prototype implementation does *not* support the atomic keyword, which would require modifying the Java compiler and/or JVM compilers to recognize atomic and transform atomic blocks to support retrying transactions and saving and restoring of local variables. Instead, our implementation requires that atomic blocks be manually transformed, as shown in Figure 4.4. This programming model involves the following transformations: (1) put the atomic block in a try–catch block to support eager aborting; (2) put that block in a do–while loop to support retrying; (3) insert calls to methods that initialize, start, and try to commit transactions; and (4) for each *local* variable used in the transaction that is reachable by a definition outside the transaction, save its old value in a "shadow" local variable before the transaction starts, and restore the old value on restart. Table 4.2 explains the APIs called by the programming model.

These transformations are all straightforward, and a production implementation could perform them automatically. Note this this programming model does *not* require manual insertion of read and write barriers, nor does it require transformations that would be difficult or impossible to perform automatically, such as manually eliding read and write barriers through knowledge of which variables are actually shared or can actually conflict.

```
1  TX.create();
2  /* Save local variables in shadow local variables */
3  do {
4    try {
5      TX.start();
6
7      /* Original code in the atomic block goes here */
8
9    } catch (RollbackException e) {
10     /* Restore local variables */
11     continue;
12   }
13 } while(!TX.tryToCommit());
```

Figure 4.4: Our prototype implementation's programming model requires manually rewriting atomic blocks to support abort and retry and to save and potentially restore local variables.

The implementation supports a configuration in which committing transactions perform "sanity checking" that checks that every object in the read/write set and undo log has an Octet state that allows the thread to read and write the object, respectively.

Eliminating and simplifying redundant barriers. Octet modifies the optimizing compiler to perform an intraprocedural, static dataflow analysis that identifies and removes redundant barriers [28]. A barrier is redundant if all incoming paths will definitely execute a barrier on the same object that is at least as "strong" (a write barrier is stronger than a read barrier) as the current barrier. LarkTM builds on this analysis to eliminate more redundant barriers. Octet's analysis cannot consider a barrier to be made redundant by a prior barrier if there is a safe point between the barriers; otherwise the safe point could respond to an access and "lose" access to an object. LarkTM safely ignores this restriction *inside transactions* since any safe point after a barrier on `o` that loses access to `o` will definitely abort. Prior work has employed similar optimizations (e.g., [76, 124]). *Outside* of

TX.create()	Indicates a thread is about to enter a new atomic region. It sets the thread's transaction timestamp to the value of a high-precision timer, the IA-32 TSC register. We implement *flattened nesting* [75], so inner transactions execute as part of the outer transaction.
TX.start()	A transaction calls this method every time it starts or restarts. TX.start() sets T.inTransaction = true, and it resets the thread's read/write set and undo log. It invokes contention management to perform exponential backoff before continuing.
TX.tryToCommit()	A transaction calls this method when it is ready to commit. It sets T.inTransaction = false, and returns true if and only if the thread's *aborting* flag has not been set. (The transaction might need to abort because of a deferred abort.) It always returns true for irrevocable transactions; it also clears the global irrevocable transaction thread.

Table 4.2: Transaction APIs.

transactions, LarkTM inserts instrumentation at barrier slow paths that "reacquires" objects that might have been acquired by other threads, allowing it to consider barriers redundant across barrier slow paths. The redundant barrier analysis is object sensitive, not field or array element sensitive, so the optimizing compiler still inserts undo log updates at each write. Identifying redundant barriers is particularly important for TM benchmarks with tight loops performing array accesses.

In non-transactional code, LarkTM-O eliminates redundant instrumentation within regions free of safe points (e.g., method calls, loop headers, and object allocations), since LarkTM's per-object biased locks ensure atomicity interrupted only at safe points. Since any lock acquire can act as a safe point, LarkTM-O adds instrumentation in non-transactional code that executes after a lock state change and reacquires any lock(s) already acquired in the current safe-point-free region, as identified by the redundant instrumentation analysis. Eliminating redundant instrumentation in non-transactional code would not guarantee

soundness for IntelSTM since it does not guarantee atomicity between safe points. However, recent work shows that statically bounded regions can be transformed to be idempotent with modest overhead [131, 50], suggesting an efficient route for eliminating redundant instrumentation. In an effort to make the comparison fair, IntelSTM eliminates instrumentation that is redundant within safe-point-free regions. LarkTM-O and IntelSTM thus use the same redundant instrumentation analysis, as does the hybrid of these two STMs, LarkTM-S.

Multi-versioning methods. LarkTM's barriers perform different behavior depending on whether they execute inside a transaction, e.g., transactional accesses behave differently from non-transactional accesses. We instruct the compiler to create different versions of compiled methods. For a method meth, we create a transactional version of the method with a prefix _atomic_, and a non-transactional version of the method with _non_atomic_. A call site invokes a different compiled version of a method depending on whether it is called from a transactional or non-transactional context. If the compiler cannot infer whether a method is called under a transactional or non-transactional context, another version of the method with an isInTransaction check for its accesses is called to check at runtime whether the access is executed under a transactional context or not.

Deferred abort. An aborting thread restarts a transaction (or a non-transactional access's barrier) by throwing a custom exception RollbackException. A requesting thread can safely throw this exception since it is executing application or library code. However, a responding thread may be executing JVM internal code, from which throwing an exception would leave the JVM in an inconsistent state. Thus, the responding thread does not abort immediately. Rather, the next transactional read or write barrier (or call to TX.tryToCommit()) will throw a

RollbackException. We make this behavior efficient by invalidating the thread's transaction identifier and RdSh hash table, so the next access will take a slow path and check the aborting flag and throw a RollbackException.

Contention management. To implement LarkTM's age-based contention management, we use IA-32's cycle counter (TSC) for timestamps. Timestamps thus do not reflect exact global ordering (providing exact global ordering could be a scalability bottleneck), but they are sufficient for ensuring progress.

Read/Write sets and undo logs. The implementation tracks read/write sets by recording information for WrEx and RdEx objects in object headers, and for RdSh objects in per-thread hash tables. We add a word to each object's header and for every static field, which is in addition to the Octet state word. This word records the last *transaction identifier* (thread identifier and transaction number) to access the object in the WrEx or RdEx state. A transaction increments its thread's transaction number every time it starts or restarts. To update the read/write set on an access of WrEx or RdEx object, the thread checks that the object's transaction identifier matches the current transaction; if not, it updates the object's transaction identifier. On a conflicting transition, the active thread checks whether the conflicting object's transaction identifier matches the responding thread's transaction, indicating a transactional conflict.

RdSh objects may be accessed by multiple transactions simultaneously. Our implementation records accesses to RdSh objects in efficient per-thread hash tables that garbage collection (GC) traces. A transaction's read of a RdSh object triggers an update to the hash table. Conflicting transitions trigger lookups in the responding thread's hash table. These

hash tables use open addressing and double hashing [45], and we build them using "raw" memory instead of pure Java objects in order to avoid extra levels of indirection.

There are other alternative data structures to build the read/write set, for example: (1) Sequential store buffer (SSB), which is fast to update, but it does not eliminate duplicates, and it requires linear time to search during conflict detection; (2) Bloom filter, which may provide faster updates [20], but it can suffer from false positive conflicts related to the so-called "birthday paradox" [166]. Furthermore, GC that moves objects effectively invalidates a bloom filter when objects are hashed based on their address.

Each thread has an undo log, implemented as a sequential store buffer (SSB). For each field or array element store, the write barrier appends three words to the undo log: the base object reference, the field offset, and the old value of the field or array element. The undo log records both the base object reference and the field offset, instead of the computed address, so GC can still trace the object reference. One bit of the offset word is reserved for indicating whether the value is a reference. For simplicity, every old value saved in the undo log is a 32-bit word, even for fields and array elements that are smaller or larger than 32 bits. For 8- and 16-bit values, the entire 32-bit aligned value that contains the old value is recorded in the undo log. For example, for a store to an element of a byte array, the undo log treats it as a store to the aligned 32-bit word that contains the byte. This simplification is correct because LarkTM guarantees that the entire object will be in the WrEx state, and objects and static fields are 32-bit aligned. For 64-bit values, the undo log records two distinct undo log entries, one for each 32-bit part of the 64-bit value.

Our implementation of read/write sets also support high-performance *generational* GC as follows. Generational GC frequently collects only recently allocated objects that reside

in a *nursery* space. To identify all transitively reachable nursery objects soundly, generational GC relies on *remembered sets* that record all references from non-nursery spaces to nursery objects. These remembered sets are updated during program execution using instrumentation at every store. Our implementation could achieve correctness by checking for references from non-nursery to nursery objects when storing references in the read-/write set and the undo log. However, this strategy would result in many remembered set entries, since the read/write set and undo log are likely to be non-nursery objects, but many program accesses are to nursery objects. Instead, we modify GC so that all collections—both nursery and full-heap collections—scan read/write sets and undo logs, avoiding the need for remembered sets for references in these data structures.

Reusable read-set. When a transaction starts (including when an existing transaction restarts), the read/write set should be cleared to avoid detecting spurious conflicts. However, we use a hash map as our implementation of the read-set for RdSh accesses, and clearing the hash table entries can be expensive, especially for short transactions.

We implement a reusable read-set that only needs to be clearly infrequently. The basic idea is to use a sliding window in an array to represent the read-set, and hashed value calculated in a previous window become obsolete once the sliding window has moved forward. Figure 4.5 gives an example. The read-set has a size of 7, and the hash function is $key\ mod\ size$. Four reads at location 16, 9, 23, and 7 in transaction Txn 42 get mapped to slots 2, 3, 4, 6. The read-set not only records the value (the location) but also the offset used to resolve the collision, e.g., location 23 is placed to slot 4 as slot 2 and slot 3 have already been occupied by prior reads 16 and 9. Once Txn 42 has finished, it moves the sliding window one slot forward. When there is a new read to location 39 from Txn 43, the

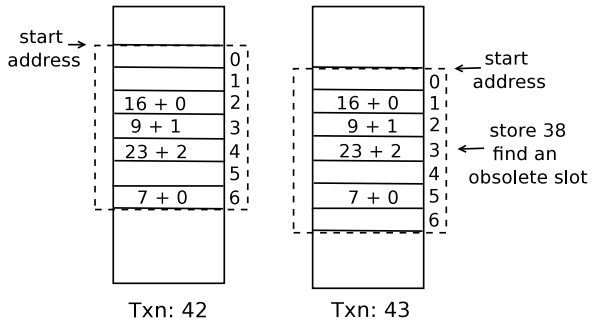


Figure 4.5: Reuse read-set to avoid overhead of clearing.

hash function decides to to put 38 to slot 3. However, it finds there is already a value 23. By calculating 23's hash location, it finds that 23 is supposed to be slot 4, which indicates that 23 is already obsolete. So the 38 can override the value 23 in slot 3.

Irrevocable transactions. Our implementation of LarkTM-O supports irrevocable transactions by letting only one transaction to exclusively become an irrevocable transaction. Before a transaction is performing a class loading or performing I/O, that transaction needs to first become irrevocable. An irrevocable transaction has the highest priority to finish and it does not follow the age-based contention management policy. Other transactions that have transactional conflicts with the irrevocable transaction, or need to perform a class loading or print must abort to let the ongoing irrevocable transaction finish first.

Zombie transactions. Our implementations of NOrec, IntelSTM, and LarkTM-S can execute zombie transactions because they validate reads lazily (Section 4.2). The implementations must perform read validation prior to commit in a few cases. (NOrec only ever needs to perform read validation if the global sequence lock has changed since the last snapshot [49].) The implementations perform read validation before throwing any runtime

exception from a transaction. The implementations mostly avoid periodic validation since infinite loops in zombie transactions mostly do not occur, except that NOrec has transactions that get stuck in infinite loops for three out of eight STAMP benchmarks. (NOrec presumably has more zombie behavior than IntelSTM since NOrec uses lazy concurrency control for both reads and writes.) For these three benchmarks only, we use a configuration of NOrec that validates reads (only if the global sequence lock has been updated) every 131,072 reads, which adds minimal overhead.

NOrec. The original NOrec design adds instrumentation after every read, which performs read validation if the global sequence lock has changed since the last snapshot [49]. This check is needed for unmanaged languages in order to avoid violating memory and type safety. Our implementation of NOrec targets managed languages, so it safely avoids this check, improving scalability (we have found) by avoiding unnecessary read validation. Our NOrec implementation can thus execute zombie transactions.

4.5 Evaluation

This section evaluates LarkTM's single-thread overhead and scalability. It evaluates both the LarkTM-O and LarkTM-S, and compares them to our implementations of IntelSTM and NOrec.

4.5.1 Experimental Setup

Section 2.4 has introduced the common experimental methodology used for evaluating all implementations. In this part, we show specific experimental setups for measuring LarkTM.

Deuce. For comparison purposes, we also evaluate the publicly available Deuce implementation [84] of the high-performance TL2 algorithm [54]. Deuce's concurrency control is at field and array element granularity, which avoids false object-level conflicts but can add instrumentation overhead. We execute Deuce with the OpenJDK JVM since Jikes RVM does not execute Deuce correctly. Evaluating Deuce helps to determine whether overhead and scalability issues are specific to our STM implementations in Jikes RVM.

Scalability. Performance shows little or no improvement beyond 8 threads, and it often degrades (anti-scales). This limitation is not unique to LarkTM or even Jikes RVM: IntelSTM and NOrec, as well as Deuce executed by OpenJDK JVM, experience the same effect. The poor scalability above 8 threads is therefore due to some combination of the benchmarks and platform. The scalability of the STAMP benchmarks is limited [167], e.g., by load imbalance and communication costs. Communication between threads executing on different 8-core processors is more expensive than intra-processor communication.

Figure 4.6 shows the scalability of two representative programs for 1–64 threads. The STM configurations generally anti-scale for 16–64 threads for kmeans_low, (which is representative of kmeans_high, ssca2, and labyrinth3d, and intruder). For vacation_low (representative of vacation_high and genome), scalability is fairly flat for 16–64 threads, with some anti-scaling.

Across all STMs we evaluate, performance is not enhanced significantly by using more than 8 threads, so our evaluation focuses on 1–8 threads (with execution limited to one 8-core processor).

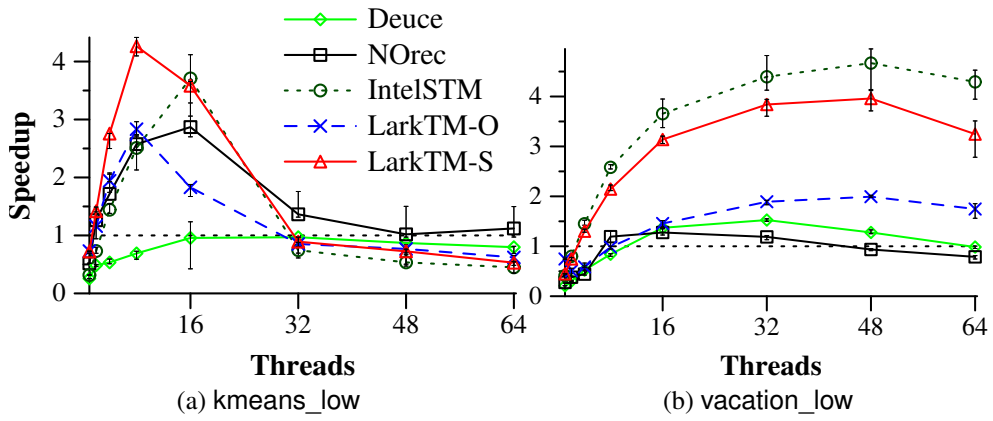


Figure 4.6: Speedup of STMs over non-STM single-thread execution for 1–64 threads for two representative programs.

Optimizations. All of our implemented STMs except NOrec perform concurrency control at object granularity, which can trigger false conflicts, particularly for large arrays divided up among threads. We refactor some STAMP benchmarks to divide large arrays into multiple smaller arrays; a production implementation could instead provide flexible metadata granularity. Furthermore, some of the most aggressive optimizations in Jikes RVM's optimizing compiler are disabled because they are buggy and perform unsound transformations, including loop unrolling and peeling, and common subexpression elimination for memory loads. These optimizations are not necessarily crucial for performance, especially on IA-32 since they can increase register pressure and lead to more register spills.

However, for several STAMP benchmarks, these optimizations are essential for allowing LarkTM to identify *redundant barriers* and thus reduce single-thread overhead. We have identified a few key hot loops across the STAMP benchmarks and performed manual transformations:

- For loops in transactions, we perform loop peeling (duplicating the the loop's body before its header), enabling redundant barrier analysis to identify accesses that are redundant across loop iterations.

- For non-transactional loops, barriers are not redundant across safe points (Section 4.4). We perform manual loop unrolling to identify some barriers redundant across loop iterations. For example, an unroll factor of four allows identifying accesses in loop iteration $4k$ that are redundant in iterations $4k + 1$, $4k + 2$, and $4k + 3$.

- For transactional and non-transactional loops, we perform redundant load elimination (RLE), common sub-expression elimination (CSE), and loop-invariant code motion (LICM). These optimizations make it possible for redundant barrier analysis to determine that two accessed objects definitely alias, e.g., two loads to a[i].f definitely access the same object.

LarkTM performs concurrency control at object granularity, which can trigger false conflicts, particularly when a program divides a large array among threads that each access different parts of the array. In the STAMP benchmarks kmeans and ssca2, different threads perform conflicting accesses to different elements of shared arrays. We refactor these programs using a transformation we call "array chunking" to eliminate this false sharing. For large arrays that experience false sharing, we divide them into multiple smaller arrays and use a level of indirection to access them. This level of indirection hurts single-thread performance—but for all configurations (the baseline, LarkTM, and IntelSTM). Future work should be able to avoid refactoring and the level of indirection by automatically assigning each "chunk" of a large array its own Octet and LarkTM metadata.

We note that all of our experiments run the manually optimized versions of the benchmarks, making the comparison fair between LarkTM, NOrec, IntelSTM, Deuce, and single-thread execution without STM support. The optimizations are straightforward and would be performed by the optimizing compiler if all of its optimizations were enabled, and by other ahead-of-time and JIT compilers, and prior work has performed these optimizations [124, 76].

Profile-guided decisions. LarkTM-S decides whether to change objects' locks to the contended state based on profiling (Section 4.2). In our experiments, LarkTM-S changes an object's lock to contended state after it performs 256 conflicting accesses. Sensitivity is low: varying the threshold from 1 to 1024 has little impact, except for kmeans, which performs worse for thresholds $\leq$128.

LarkTM-S uses offline profiling to select types (Java classes) whose instances should be locked into contended state at allocation time. The policy selects types whose ratio of conflicting to non-conflicting accesses is greater than 0.01, excluding common types such as int arrays and Object. It limits the selected types so that at most 25% of the execution's accesses are to contended objects, since otherwise the execution might as well use IntelSTM instead of LarkTM-S. Since profiling and performance runs use the same inputs, they represent a best case for online profiling.

4.5.2 Execution Characteristics

Table 4.3 reports instrumented accesses executed by the four implemented STMs during single-thread execution. The table shows that while reads outnumber writes, writes are not uncommon. Several programs spend almost all of their time in transactions, while a few spend significant time executing non-transactional accesses. NOrec instruments more

	NOrec			IntelSTM, LarkTM-O, and LarkTM-S				
	Total	Transactional		Total	Transactional		Non-transactional	
	accesses	reads	writes	accesses	reads	writes	reads	writes
kmeans_low	1.0×10^9	7.0×10^8	3.5×10^8	7.2×10^9	3.4×10^7	1.3×10^7	7.1×10^9	2.7×10^7
kmeans_high	1.4×10^9	9.2×10^8	4.6×10^8	7.5×10^9	2.4×10^7	9.1×10^6	7.4×10^9	4.6×10^7
ssca2	4.6×10^7	3.5×10^7	1.2×10^7	4.5×10^9	3.4×10^7	1.2×10^7	3.5×10^9	4.2×10^8
intruder	1.5×10^9	1.4×10^9	1.0×10^8	8.8×10^8	7.2×10^8	6.0×10^7	5.4×10^7	5.3×10^4
labyrinth3d	7.2×10^8	6.8×10^8	4.6×10^7	4.1×10^8	3.5×10^8	4.6×10^7	1.9×10^3	5.4×10^2
genome	1.7×10^9	1.7×10^9	6.7×10^7	5.3×10^8	2.9×10^8	6.9×10^5	2.1×10^8	2.1×10^6
vacation_low	1.4×10^9	1.3×10^9	7.8×10^7	7.9×10^8	7.2×10^8	2.9×10^7	2.0×10^3	1.3×10^7
vacation_high	1.9×10^9	1.8×10^9	1.0×10^8	1.1×10^9	1.0×10^9	4.0×10^7	1.1×10^4	2.1×10^7

Table 4.3: Accesses instrumented by NOrec, IntelSTM, LarkTM-O, and LarkTM-S during single-thread execution.

	LarkTM-O		LarkTM-S			
	Same state	Conflicting	Same state	Conflicting	Contended read	Contended write
kmeans_low	6.3×10^9 (**99.60**%)	1.3×10^7 (**0.20**%)	6.2×10^9 (**99.49**%)	8.7×10^4 (**0.0014**%)	1.6×10^7 (**0.25**%)	1.6×10^7 (**0.25**%)
kmeans_high	7.6×10^9 (**99.69**%)	1.2×10^7 (**0.16**%)	7.6×10^9 (**99.65**%)	8.2×10^4 (**0.0011**%)	1.3×10^7 (**0.17**%)	1.3×10^7 (**0.17**%)
ssca2	6.5×10^9 (**99.71**%)	1.2×10^7 (**0.19**%)	5.3×10^9 (**98.0**%)	5.8×10^6 (**0.11**%)	9.0×10^7 (**1.7**%)	9.2×10^6 (**0.18**%)
intruder	1.4×10^9 (**91.6**%)	6.3×10^7 (**4.3**%)	1.1×10^9 (**76**%)	3.9×10^7 (**2.7**%)	2.6×10^8 (**11**%)	2.0×10^7 (**1.4**%)
labyrinth3d	4.6×10^8 (**99.9910**%)	2.2×10^4 (**0.0048**%)	4.5×10^8 (**99.997**%)	2.2×10^4 (**0.0048**%)	9.5×10^2 (**0.00021**%)	1.3×10^2 (**0.000028**%)
genome	6.8×10^8 (**97.1**%)	1.8×10^7 (**2.6**%)	4.5×10^8 (**79**%)	1.2×10^5 (**0.021**%)	8.2×10^7 (**14**%)	2.1×10^6 (**0.37**%)
vacation_low	7.8×10^8 (**94.3**%)	2.7×10^7 (**3.3**%)	7.2×10^8 (**81**%)	2.4×10^6 (**0.27**%)	1.4×10^8 (**9.9**%)	1.7×10^7 (**1.9**%)
vacation_high	1.1×10^9 (**95.0**%)	3.2×10^7 (**2.8**%)	9.7×10^8 (**78**%)	2.5×10^6 (**0.20**%)	2.5×10^8 (**13**%)	2.1×10^7 (**1.7**%)

Table 4.4: Lock acquisitions when running LarkTM-O and LarkTM-S. *Same state* accesses do not change the lock's state. *Conflicting* accesses trigger the coordination protocol and conflict detection. *Contended state* accesses use IntelSTM's concurrency control. Percentages are out of total instrumented accesses (unaccounted-for percentages are for upgrading lock transitions). Each percentage x is rounded so x and $100\% - x$ have at least two significant digits.

transactional accesses than the other STMs because it cannot exclude instrumentation from redundant accesses (Section 4.4). *Transactional writes* does not count the undo log instrumentation that IntelSTM, LarkTM-O, and LarkTM-S add at every transactional write (Section 4.4).

Table 4.4 reports lock state transitions for LarkTM-O and LarkTM-S running STAMP with 8 application threads. The *Same state* column reports how many instrumented accesses require no lock state change, meaning they take the fast path. For LarkTM-O, more

than 90% of accesses fall into this category for every program. *Conflicting* lock acquires require coordination with other thread(s) in order to change the lock's state. Although LarkTM-O achieves a relatively low fraction of lock acquires that are conflicting—always less than 5%—coordination costs affect scalability significantly.

LarkTM-S successfully avoids many conflicting transitions by using the contended state, often reducing conflicting lock acquires by an order of magnitude or more. At the same time, many same-state accesses become contended-state accesses. More than 10% of accesses are to contended objects in four programs (intruder, genome, vacation_low, and vacation_high).

Table 4.5 counts transactions committed and aborted for the four STMs implemented in Jikes RVM, running STAMP with 8 threads. Different conflict resolution and contention management policies lead to different abort rates for the STMs. Several programs have a trivial abort rate; others abort roughly 10% of their transactions. LarkTM-O and LarkTM-S have different abort rates because LarkTM-S uses IntelSTM's conflict resolution and contention management for contended accesses. Although we might expect IntelSTM's suboptimal contention management to lead to more aborts, the implementations are not comparable: LarkTM always resolves conflicts by aborting a thread, while IntelSTM waits for some time (rather than aborting immediately) for a contended lock to become available. NOrec often has the lowest abort rate, mainly (we believe) because it performs conflict detection at field and array element granularity, so its transactions do not abort due to false sharing. In contrast, the other STMs detect conflicts at object granularity. As our performance results show, *abort rates alone do not predict scalability*, which is influenced strongly by other factors such as LarkTM's coordination protocol and NOrec's global lock.

	Transactions committed	Transactions aborted at least once			
		NOrec	IntelSTM	LarkTM-O	LarkTM-S
kmeans_low	6.2×10^6	4.4%	0.2%	1.8%	0.2%
kmeans_high	5.1×10^6	3.7%	0.3%	2.9%	0.4%
ssca2	5.8×10^6	$< 0.1\%$	4.1%	4.7%	2.8%
intruder	2.4×10^7	7.5%	24.2%	35.1%	7.9%
labyrinth3d	2.9×10^2	3.8%	15.3%	0.3%	$< 0.1\%$
genome	2.5×10^6	$< 0.1\%$	0.1%	0.2%	$< 0.1\%$
vacation_low	4.2×10^6	$< 0.1\%$	0.3%	8.4%	0.1%
vacation_high	4.2×10^6	$< 0.1\%$	0.5%	7.6%	$< 0.1\%$

Table 4.5: Transactions committed and aborted at least once for four STMs.

4.5.3 Performance Results

This section compares the performance of the STMs with each other and with uninstrumented, single-thread execution.

Overhead of strong atomicity. We first evaluate the cost of strong atomicity alone by using *non-transactional* benchmarks: the DaCapo Benchmarks [18], versions 2006-10-MR2 and 9.12-bach (excluding benchmarks that are single-threaded or that Jikes RVM cannot run[12]) with the large workload size; and fixed-workload versions of SPECjbb2000 and SPECjbb2005.[13]

Figure 4.7 presents the runtime overhead IntelSTM, LarkTM-O, and LarkTM-S add to non-transactional programs. (Deuce and NOrec are weakly atomic; they do not instrument non-transactional accesses.) IntelSTM, which performs an atomic operation at every non-transactional write and validates every non-transactional read, slows non-transactional programs by 75% on average. LarkTM-O and LarkTM-S incur coordination overhead due

[12]We also exclude avrora9 since our implementation of IntelSTM fails on it for a reason we have not yet diagnosed, although LarkTM runs it correctly.

[13]`http://spec.org/jbb200{0,5}`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

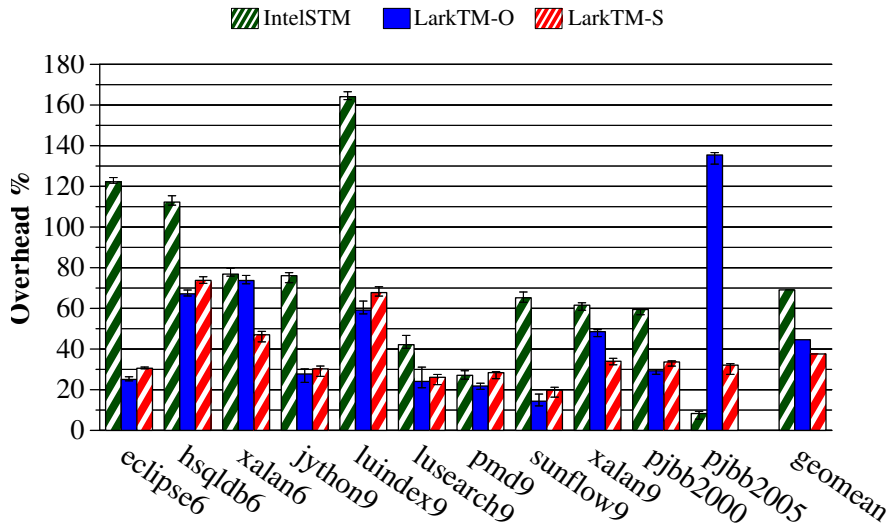


Figure 4.7: Overhead added to *non-transactional* programs by strongly atomic STMs. The `6` and `9` suffixes distinguish DaCapo 2006 and 2009.

to conflicting accesses, since these programs are multi-threaded. This effect is greatest for LarkTM-O running `pjbb2005`, which has the highest rate of conflicting accesses; LarkTM-S reduces this overhead substantially by putting high-conflict objects' locks into the contended state. On average, LarkTM-O and LarkTM-S provide strong atomicity with 41 and 33% overhead, respectively. For this experiment, LarkTM-S uses only online object-based (not offline type-based) profiling.

Single-thread overhead. Transactional programs execute multiple parallel threads in order to achieve high performance. Nonetheless, single-thread overhead is important because it is the *starting point* for scaling performance with more threads. Existing STMs have struggled to achieve good performance largely because of high instrumentation overhead (Section 1.1.2) [160, 37].

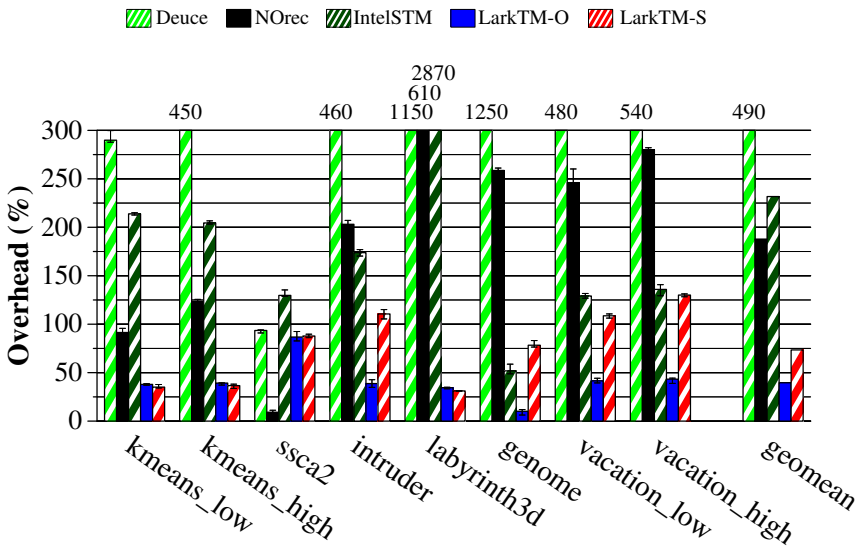


Figure 4.8: Single-thread overhead (over non-STM execution) added by the five STMs. Lower is better.

Figure 4.8 shows the single-thread overhead (i.e., instrumentation overhead) of the five STMs, compared to single-thread performance on Jikes RVM without STM, except for Deuce, which is normalized to single-thread performance on OpenJDK JVM. Deuce slows programs by almost 6X on average relative to baseline OpenJDK JVM, which we find is 33% faster than Jikes RVM on average.

Our NOrec and IntelSTM implementations slow single-thread execution significantly—by 2.9 and 3.3X on average—despite targeting low overhead. NOrec in particular aims for low overhead and reports being one of the lowest-overhead STMs [49]. IntelSTM targets low overhead by combining eager concurrency control for writes with lazy read validation [135]. Yet they still incur significant costs: NOrec buffers each write; and it looks up each read in the write set and (if not found) logs the read in the read validation log. IntelSTM performs atomic operations at many writes, and it logs and later validates

reads. LarkTM-O yields the lowest instrumentation overhead (1.40X on average), since it minimizes instrumentation complexity at non-conflicting accesses. LarkTM-S's single-thread slowdown is 1.73X; its instrumentation uses atomic operations and read validation for accesses to objects with locks in contended state. In single-thread execution, LarkTM-S puts objects into contended state based on offline type-based profiling only.

An outlier is ssca2, for which NOrec performs the best, since a high fraction of its accesses are non-transactional (Table 4.3). While kmeans_low and kmeans_high also have many non-transactional accesses, the overhead of its transactional accesses, which execute in relatively short transactions, is dominant.

IntelSTM's very high overhead on labyrinth3d is related to its long transactions, which lead to large read and write sets. IntelSTM's algorithm has to validate some read set entries by linearly searching the (duplicate-free) write sets, adding substantial overhead for labyrinth3d because its write sets are often large. IntelSTM could avoid this linear search by incurring more overhead in the common case, as in a related design [76]. If we remove the validation check, IntelSTM still slows labyrinth3d's single-thread execution by 4X.

NOrec also adds high overhead for labyrinth3d. We find that whenever the instrumentation at a read looks up the value in the write set, the average write set size is about 64,000 elements. In contrast, the average write set size is at most 16 elements for any other program. Although our NOrec implementation uses a hash table for the write set, it is plausible that larger sizes lead to more-expensive lookups (e.g., more operations and cache pressure).

Scalability. Figure 4.9 shows speedups for the STMs over non-STM single-thread execution for 1–8 threads. Each single-thread speedup is simply the inverse of the overhead from Figure 4.8.

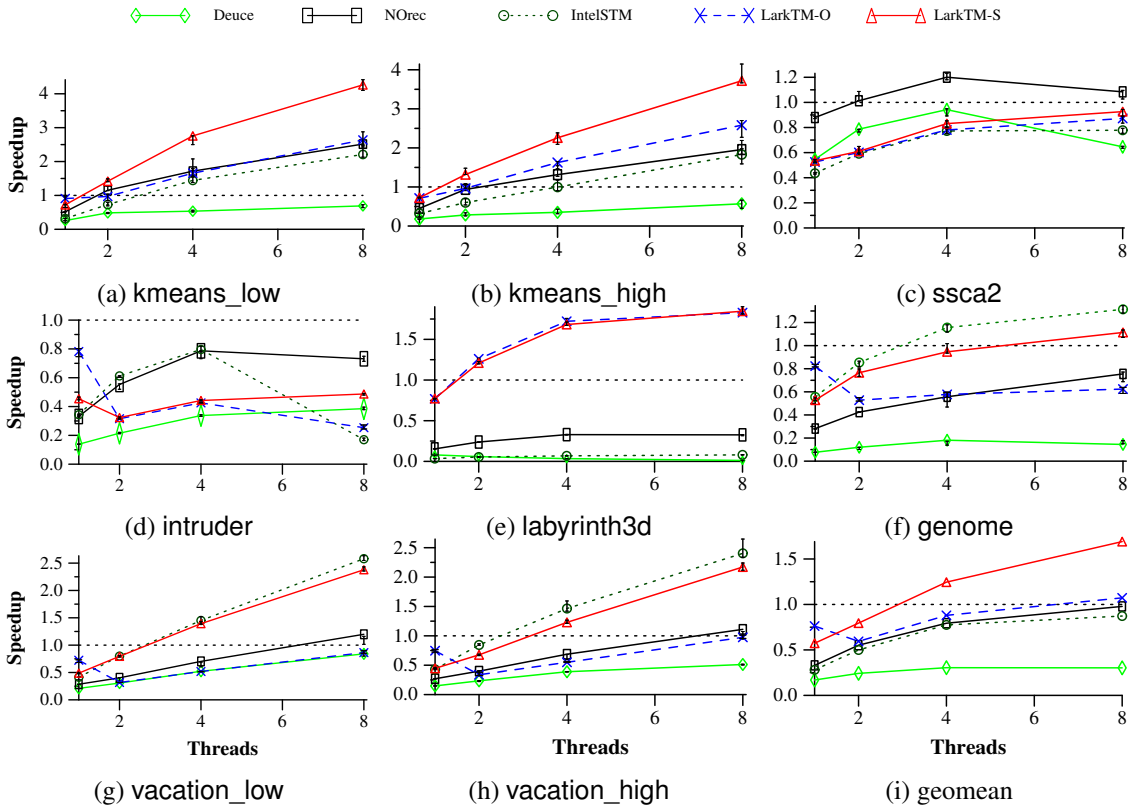


Figure 4.9: Performance of Deuce, NOrec, IntelSTM, LarkTM-O, and LarkTM-S, normalized to non-STM single-thread execution (also indicated with a horizontal dashed line). The x-axis is the number of application threads. Higher is better.

Deuce, NOrec, and IntelSTM scale reasonably well overall, but they start from high single-thread overhead, limiting their overall best performance (usually at 8 threads). LarkTM-O has the lowest single-thread overhead on average, yet it scales poorly for several programs that have a high fraction of accesses that trigger conflicting transitions—particularly genome and intruder. Execution time increases for vacation_low and vacation_high from 1 to 2 threads because of the cost of coordination caused by conflicting lock acquires, then

decreases after adding more threads and gaining the benefits of parallelism. LarkTM-S achieves scalability approaching IntelSTM's scalability because LarkTM-S effectively eliminates most conflicting lock acquires. Starting at two threads, LarkTM-S provides the best average performance by avoiding most of LarkTM-O's coordination costs while retaining most of its low-cost instrumentation benefits.

Just as prior STMs have struggled to outperform single-thread execution [55, 37, 160, 3], Deuce, NOrec, and IntelSTM are unable, on average, to outperform non-STM single-thread execution. In contrast, LarkTM-O and LarkTM-S are 1.07X and 1.69X faster, respectively, than (non-STM) single-thread execution.

Figure 4.9(i) shows the geomean of speedups across benchmarks. The following table summarizes how much faster LarkTM-O and LarkTM-S are than other STMs:

	Deuce	NOrec	NOrec$^-$	IntelSTM	IntelSTM$^-$
LarkTM-O	3.54X	1.09X	0.93X	1.22X	0.87X
LarkTM-S	5.58X	1.72X	1.47X	1.93X	1.37X

The numbers represent the ratio of LarkTM-O or LarkTM-S's speedup to each other STM's speedup, all running 8 threads. *NOrec*$^-$ and *IntelSTM*$^-$ are geomeans without labyrinth3d.

Results on a different platform. We have repeated the performance experiments on a system with four Intel Xeon E5-4620 8-core processors (32 cores total).

Figure 4.10 shows speedups for each STAMP benchmark and the geomean. Single-thread overhead and scalability are similar across both platforms. As on the AMD platform, NOrec, IntelSTM, and LarkTM-O have similar performance on average on the Intel platform, although LarkTM-O performs slightly worse in comparison on the Intel platform. On both platforms, LarkTM-S significantly outperforms the other STMs on average.

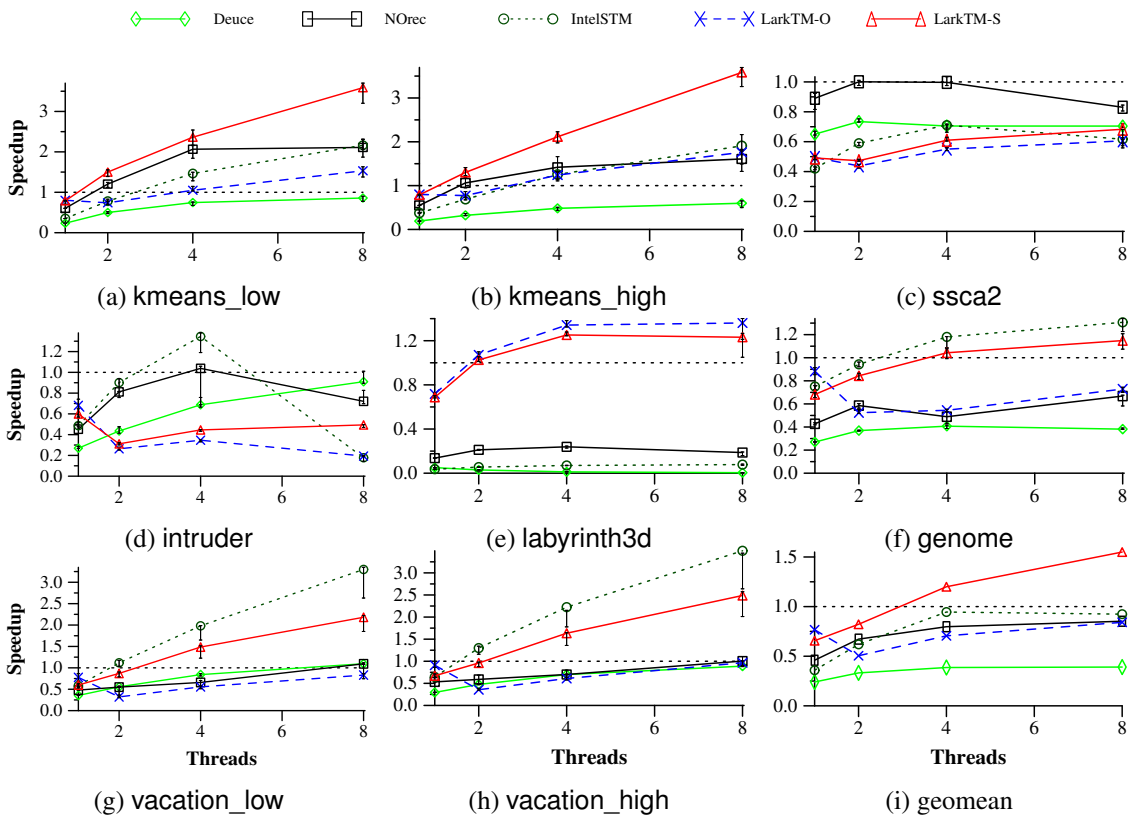


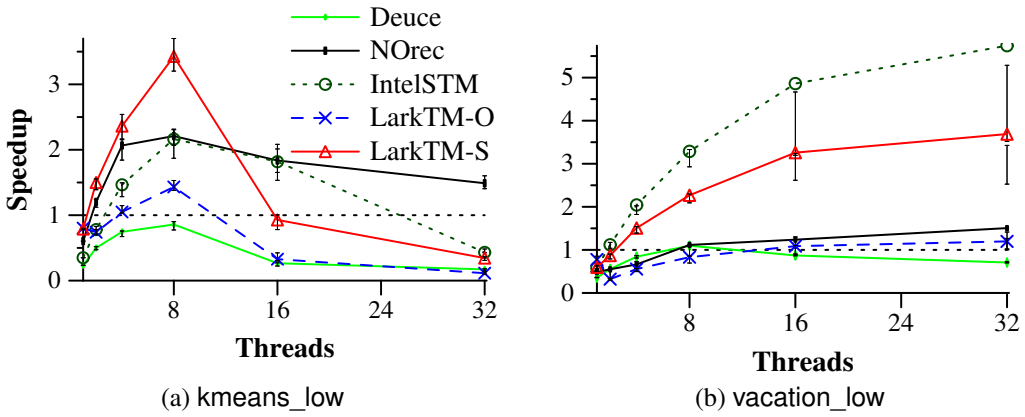
Figure 4.10: STM performance for 1–8 threads on an Intel Xeon platform. Otherwise same as Figure 4.9.

Figure 4.11 shows scalability for 1–32 threads for the same two representative STAMP benchmarks as Figure 4.6. Although on vacation_low the STMs may seem to scale better on the Intel machine, we note that Figure 4.11 evaluates only 1–32 threads.

Summary. Across all programs, LarkTM-O provides the lowest single-thread overhead, NOrec and IntelSTM typically scale best, and LarkTM-S does well at both.

4.6 Conclusion and Interpretation

As introduced in Section 1.1, runtime support can be used to provide high-level programming abstractions to better express parallelism. One such an example is transactional



(a) kmeans_low (b) vacation_low

Figure 4.11: STM performance for 1–32 threads on an Intel Xeon platform. Otherwise same as Figure 4.6.

memory, which serves as a programming model that simplifies concurrent programming. The dissertation identifies the main challenges of existing STMs: either they suffer from high overhead or support only weak semantics. To make transactional memory more practical, this chapter presents two novel STM designs that provide high performance and strong semantics.

To tackle one of STM's key cost in existing work—detecting and handling conflicting accesses, we introduce LarkTM-O, which uses a novel mechanism for tracking conflicting dependences that avoids synchronization except at truly conflicting accesses. LarkTM-O provides strong atomicity with low overhead. Furthermore, to address how to make our approach scale to high-contention workloads, we introduce a hybrid mechanism LarkTM-S, which uses online profiling to identify *contended* objects and dynamically switches those objects to use less-aggressive concurrency control that incurs less synchronization overhead (but slightly more instrumentation overhead) for high-contention accesses. The hybrid approach achieves the best of both worlds in terms of performance and scalability. Our

performance results show that both LarkTM-O and LarkTM-S have much lower overhead and high scalability than state-of-the-art STMs. Furthermore, LarkTM provides strong semantics and strong progress guarantees, justifying our choice of such strong semantics over weak semantics provided by many existing STMs.

By showing how to build STM systems with low overhead and strong semantics, this work can promote the use of transactional memory in mainstream concurrent programming. Our work would help researchers and practitioners to evaluate STM's true programmability and reliability benefits. We believe that widespread use of STM would encourage more commercial development of hardware support for TM (e.g., to build hardware–software hybrid TMs, which still need STM support). Practical TM would also provide system benefits such as protection against security attacks, which would make concurrent programs more reliable and scalable, enabling greater reliance on concurrent programs for safety- and mission-critical tasks.

Chapter 5: Software Transactional Memory with Relaxed Dependence Tracking

Chapter 4 shows that STM can support strong semantics and strong progress guarantees in an efficient, scalable way on existing multicore processors. Chapter 3 introduces the relaxed dependence tracking that can hide coordination latency. Is it possible to combine the features of both LarkTM and RT: support for low-overhead and strongly atomic STM with even better scalability? This chapter introduces *RT-based STM*, which extends from both LarkTM-O and RT to hide latencies caused by transactional conflict detection. We also discuss how the resulting composite implementation compares with the existing STM.

5.1 Extending LarkTM-O with RT

This section describes how we extend LarkTM-O (introduced in Section 4.1) to use RT. LarkTM-O uses biased reader–writer locks. For ease of description, we denote biased reader–writer locks as strict dependence tracking since it tracks each cross-thread dependences soundly (i.e., does not miss any dependences). We refer LarkTM-O as *ST-based STM* since LarkTM-O employs biased reader–writer locks to provide eager concurrency control: it detects and resolves conflicts before performing each memory access. Conflict detection and resolution piggyback on coordination.

Here we focus on how the STM piggybacks on coordination that uses an *explicit* request. In that case, the *responding* thread detects and resolves conflicts between the responding thread's transaction and the requesting thread's transaction or non-transactional access.

Extending the LarkTM-O to use RT presents challenges. Unless handled properly, the STM could be unable to detect and resolve transactional conflicts for relaxed loads and stores. Figure 5.1(a) shows an example of a problematic execution. Thread T2 performs two relaxed loads from o.f in a transaction, since o's lock is in $\mathsf{WrEx}_{\mathsf{T1}}$ state. T1 performs conflict detection when it responds to T2, but by then T1 has started another transaction that has not accessed o, so T1 accurately reports no transactional conflict. Similarly, T1's transaction might have accessed o, but once T1 receives the request, it has already left the transaction, causing T1 to report no transactional conflict either (Figure 5.1(b)). However, the result is unserializable because T2's loads see different values. Another problematic issue (not shown) is that performing relaxed transactional stores directly could lead to unserializable results due to another thread loading the value simultaneously.

Our RT-based STM addresses these issues as follows. In RT-based STM, each relaxed, transactional load logs its loaded value, and *validates* the value later. (In Figure 5.1, T2's transaction would fail the read validation before commit and abort.) Each relaxed, transactional store is deferred, to provide opacity of intermediate updates before the transaction commits. Before a transaction commits, it waits for coordination responses so it can validate all relaxed loads and perform all deferred stores.

Relaxed loads. A requesting thread reqT performs a relaxed load when it reads from an object o locked in the $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ or $\mathsf{RdEx}^{\mathsf{Int}}_{\mathsf{reqT}}$ state. reqT first checks its store buffer for the

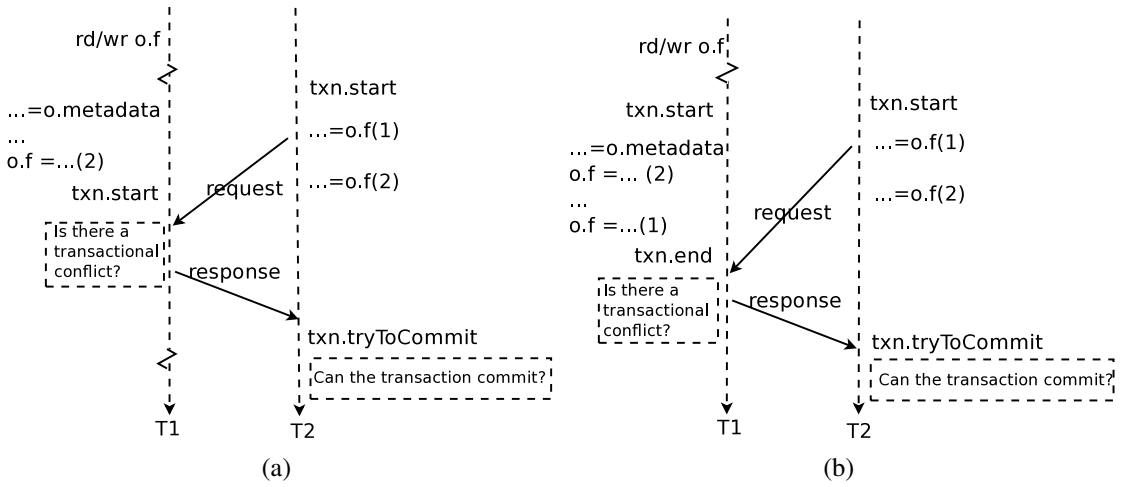


Figure 5.1: Allowing unhandled relaxed accesses in transactions would lead to serializability violations. The values in parentheses after each executed store and load are the values written and read, respectively.

value (only if the object's state is $\mathsf{WrEx}^{\mathsf{Int}}_{\mathsf{reqT}}$). If not found, reqT performs the load—but responding thread(s) may be simultaneously writing to o, potentially violating serializability. reqT thus logs the loaded location and value in a *read validation log*.

When reqT receives the response for o, it validates all entries in the read validation log against o's *current* value(s), as the following pseudocode shows:

```
foreach (addr, value) in readValidationLog
  if (*addr != value) abortTxn();
```

For every field or array element of o in the read validation log, the current value must match the log's value. This logic makes sense as follows. The responding thread responded at some safe point where it performed conflict detection (and potentially conflict resolution). Validating the loaded value ensures that the values that were read previously for o are the same as if the values had all been read at the responding thread's responding safe point.

If validation fails, reqT must abort its current transaction. (If respT responds implicitly, it performs the above work on behalf of reqT.)

Relaxed stores. A requesting thread reqT defers a relaxed store by buffering its location (address) and value in the store buffer, which is analogous to the *redo log* used by STMs that use lazy versioning [75]. After reqT receives all responses for o, it performs the store(s) for o from the store buffer—and also logs the store(s) in the *undo* log. (If respT responds implicitly, it performs all of these actions on behalf of reqT.)

Commit and abort. Before a transaction commits or aborts, it waits for all outstanding responses, in order to validate loads and perform deferred stores. Unlike our general RT design, our RT-based STM does *not* support loads by T to objects locked in intermediate states *other than* $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Int}}$ and $\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Int}}$; supporting loads from other states would require a mechanism for eventually changing the lock's state to $\mathsf{WrEx}_{\mathsf{T}}$, $\mathsf{RdEx}_{\mathsf{T}}$, or RdSh in order to validate reads before commit.

5.2 Comparison with ST-based STM

This section compares the progress guarantees and semantics of RT-based STM against that of ST-based STM.

Guaranteeing progress. The *ST-based* STM guarantees progress by detecting all conflicts eagerly and then aborting the younger transaction [141, 164]. However, the *RT-based* STM cannot guarantee progress, since any transaction that fails read validation must abort. Other mixed-mode STMs have similarly lacked progress guarantees [76, 124]. Standard techniques such as exponential backoff can help to alleviate livelock. For RT-based STM,

a simple (unimplemented) solution exists: if a transaction repeatedly fails read validation, it falls back to use *strict* dependence tracking, guaranteeing it will commit (at least once it becomes oldest).

Correctness. At a high level, RT-based STM provides serializability by guaranteeing that all of a committing transaction's operations appear as though they happened instantaneously at commit time. For conflicting accesses handled by ST, eager conflict detection and resolution guarantee that conflicting accesses between the committing transaction's accesses and commit time will be detected and resolved. (RT-based STM uses the same mechanism for relaxed stores, but defers making the store visible until relaxed coordination has finished.) For relaxed loads, commit-time value validation ensures that each value from a relaxed load is consistent with the commit-time value of the memory location.

Semantics. Lazy read validation can lead to so-called *zombie* transactions whose behavior is impossible in any serializable execution [75]. In managed languages such as Java, zombies are not a serious problem because memory and type safety are preserved [104, 47]. Targeting a native language such as C++ would require additional support to provide *sandboxing* of zombie transactions [47]. Another issue is that zombie transactions can get stuck in infinite loops that are impossible in any serializable execution. RT-based STM cannot experience this issue because it validates relaxed loads within a bounded amount of time.

5.3 Evaluation

We compare the ST-based STM and RT-based STM using the transactional STAMP benchmarks. STAMP provides low- and high-contention workloads for kmeans and vacation; here we evaluate only low contention because high contention shows virtually identical differences between the two STMs. Figure 5.2 shows the execution time of the ST-based STM and RT-based STM. We first note that both STMs typically scale poorly after 8 threads; prior work has also found that STAMP has limited scalability [167]. Furthermore, our platform has 8 cores per socket, leading to greater inter-thread communication for >8 threads.

For genome and vacation, RT reduces overhead for 2–8 application threads, but the benefit decreases with more threads. For genome, the ratio of implicit to explicit requests increases substantially with more threads (statistics not shown), leading to fewer opportunities for RT to improve performance. For vacation, the implicit-to-explicit ratio stays fairly constant across thread counts, but RT's benefit diminishes because accesses per thread decrease as threads increase, leading to less latency per thread for RT to reduce.

Fewer than 0.01% of labyrinth3d's accesses trigger coordination, so it cannot benefit noticeably from RT.

For kmeans, intruder, and ssca2, RT provides sustained or increasing benefit over ST for 8 to 32 threads. For these programs, RT-based STM achieves a significantly lower rate of aborting transactions than the ST-based STM. RT-based STM validates relaxed loads at object field and array element granularity, as opposed to the ST-based STM's loads, which use reader locks and read sets at object granularity, leading to more transactional conflicts due to false sharing—an interesting side effect of supporting relaxed loads. However, direct

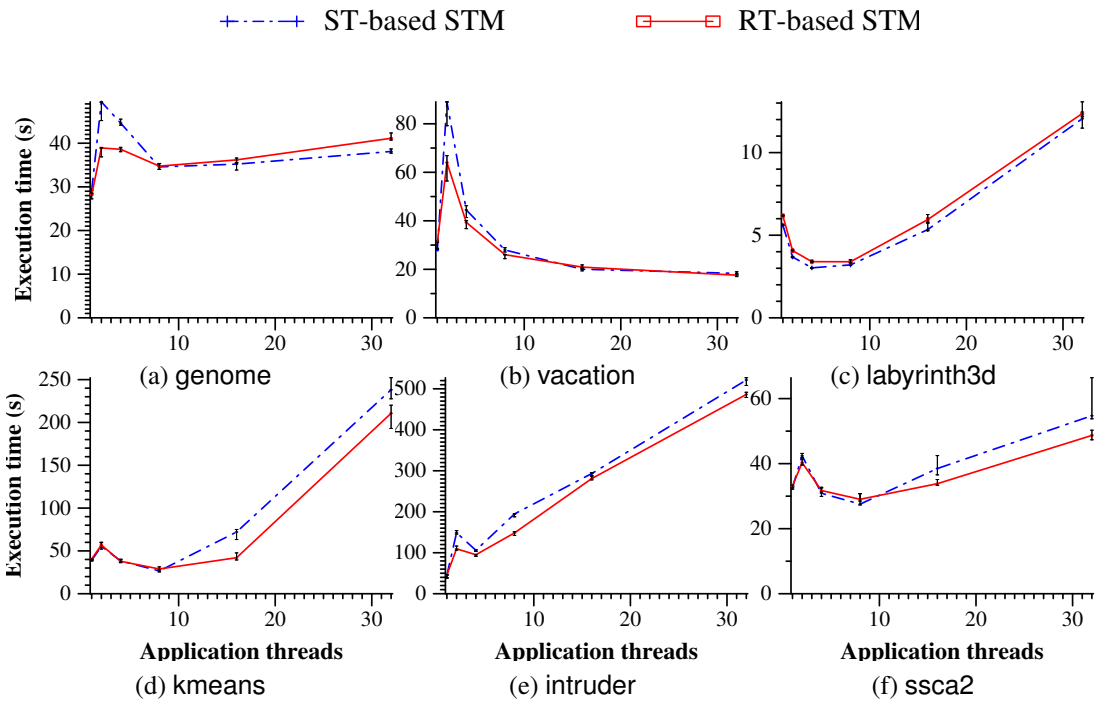


Figure 5.2: Performance of ST-based STM and RT-based STM.

benefits from RT are limited because many transactions are short, and RT-based STM must wait at transaction end for all outstanding relaxed accesses (Section 3.2).

5.4 Conclusion and Interpretation

In Chapter 3, we introduced RT and we claimed it could be used for supporting many runtime support, and in Chapter 4 we introduced a runtime support based STM system LarkTM. In this chapter, we demonstrated RT's practicality by extending LarkTM-O with RT. The extended STM scales better than LarkTM-O while still permitting strong atomicity and progress guarantees. Tests show that this approach is a viable option even though RT's benefit is somewhat limited by TM's correctness constraints. Ultimately, this work shows

the possibilities to further improve the scalability of dependence-tracking-based runtime support while still being able to preserve low-overhead and strong semantics.

Chapter 6: Strong Memory Consistency Models With Fail-Stop Semantics

Allowing multiple threads to concurrently read and write a set of common memory locations complicates the behavior or semantics of memory operations in a shared-memory platform. However, programmers need a conceptual model (i.e., referred to as *memory model* or *memory consistency model*) for the semantics of memory operations to allow them to correctly write concurrent programs. The memory model therefore is at the heart of the concurrency semantics. Modern compilers perform optimizations such as code motion, loop transformations, and register allocation, and hardware performs optimizations such as pipelining, branch prediction, forwarding, issuing multiple instructions in the same clock cycle(s), and even reordering the instruction stream for out-of-order execution. Most of these optimizations are safe for sequential programs, but can change the meaning of concurrent programs and cause visible effects that can break programmer expectations. A memory model defines the interface between the program and any hardware or software that may transform that program, trying to eliminate the gap between the behavior expected by the programmer and the actual behavior supported by a system.

Modern weak consistency model such as Java memory model (JMM) and C++ memory model provide only guarantees to data-race-free programs [100, 4]. Researchers therefore have also proposed stronger memory consistency models that treat data races as legal but

exceptional behavior and throw an exception in the presence of a data race so that a data race can still have well-defined behavior (as introduced in Section 2.3). Existing strong memory models are promising, but has some potential problems: mainly exceptions and complexity.

This chapter introduces a new strong memory model and a mechanism on how to enforce the strong memory model in runtime. It is also the first to our knowledge to consider and address the problem of availability in memory consistency models that generate consistency exceptions. The goals are to explore alternatives to current approaches that provide the strong memory model RSx (*region serializability exceptions*, see Section 2.3) and to develop approaches that provide better performance–availability tradeoffs than existing approaches.

Section 6.2 develops mechanisms for avoiding consistency exceptions under RSx, which improve availability significantly. In an effort to improve availability further, Section 6.3 introduces a new memory model called SIx, which is still *principled*, meaning that it provides well-defined semantics at region granularity, and does not inhibit compiler and hardware optimizations. Section 6.4 presents two approaches for providing SIx that represent different points in the performance–availability space. Section 6.7 evaluates the availability, performance, scalability, space overhead, and runtime characteristics of our mechanisms to support strong memory models.

6.1 Motivation

In section 2.3, we introduced a memory model RSx. Although RSx provides strong, well-defined semantics, supporting it incurs a major disadvantage: the possibility of consistency exceptions. Another challenge is the cost and complexity of detecting region conflicts. The drawbacks of providing RSx are:

Availability. RSx provides well-defined semantics even for racy executions. On the other hand, any racy execution can throw a consistency exception, essentially trading availability for strong, well-defined semantics. Data races occur unexpectedly, and then may or may not manifest unexpectedly as consistency exceptions depending on whether the race manifests as a region conflict.

Under RSx, data races are potential fail-stop errors, just like buffer overflows and null pointer exceptions in memory- and type-safe languages such as Java. Ideally, programmers would eliminate nearly all data races by addressing consistency exceptions encountered during testing and early production runs (e.g., alpha and beta testing). An additional mitigating factor is that programmers or automatic techniques may be able to *handle* consistency exceptions in a way that preserves availability. Nonetheless, we expect that consistency exceptions will occur unexpectedly and affect availability—just like null pointer exceptions—which may frustrate developers and users *more than* the (often silent and unknown) consequences of racy executions under DRF0. Prior work on memory models that generate consistency exceptions has not considered the issue of availability nor how to reduce exceptions [58, 17, 98, 101, 137].

Performance. Existing work that provides RSx incurs significant cost and complexity, whether implemented in hardware or software. *Conflict Exceptions* augments the cache coherence protocol in order to detect region conflicts, and adds on-chip network traffic and space overheads, making cache evictions and region boundaries more expensive [98]. Biswas et al. present two software-only approaches for providing RSx [17]. The first, *FastRCD*, slows down executions by 3.7X on average. The second approach, *Valor*, slows executions by 2.0X on average, through lazy detection of read–write conflicts. While advantageous for performance, it cannot tolerate read–write conflicts leading to consistency exceptions, although we find this problem is not substantial in practice on average. Valor incurs two other disadvantages: it provides asynchronous exceptions and cannot easily handle unsafe languages such as C++ (Section 6.2.2). Common to existing software and hardware approaches for providing RSx is the cost and complexity of tracking the last region(s) to *read* each variable (for example, Valor has to log reads [17]), in order to detect or infer read–write conflicts accurately.

6.2 Increasing Availability Under RSx

This section extends two analyses from prior work [17] that provide RSx, *FastRCD* and *Valor*, in order to avoid consistency exceptions while still providing RSx. While these extensions by themselves are not huge intellectual contributions, our work is the first to introduce and evaluate them.

6.2.1 FastRCD and FastRCD-A

FastRCD is an analysis from prior work that provides RSx. Our presentation of FastRCD, including notations and algorithms, is closely based on prior work's presentation [17].

We extend FastRCD to *wait*, instead of throwing consistency exception, when it detects a region conflict. We call the resulting analysis *FastRCD-A* (Available).

FastRCD uses *epoch optimizations* from an existing data race detection analysis called *FastTrack* [63]. The analysis maintains, for each shared variable, (1) the last region that wrote the variable and (2) the last region(s) (one per thread) that read the variable since the last write. The analysis identifies regions by maintaining a per-thread logical clock for each thread; a thread's clock starts at 1, and the analysis increments it every time the thread executes a region boundary. FastRCD uses the following notation:

clock(T) – Returns the current clock c of thread T.

epoch(T) – Returns the epoch c@T, where c is the current clock of thread T.

$\mathcal{W}_x$ – Represents last-writer metadata for variable x, as the epoch c@t, which means t last wrote x at time c.

$\mathcal{R}_x$ – Represents last-reader metadata for x, as a *read map* that maps each thread to a clock value (or 0 if not present in the map).

Our FastRCD-A analysis extends FastRCD by *waiting* at detected region conflicts, until either (1) the region conflict no longer exists, in which case the analysis proceeds, or (2) the analysis detects a cyclic waiting dependency, in which case the analysis throws a consistency exception. Algorithms 1, 2, and 3, show FastRCD-A's analysis at region boundaries and program writes and reads, respectively.

Next, we describe the high-level operation of FastRCD-A's (and FastRCD's) analysis at program reads and writes. The first **if** statement in Algorithms 2 and 3 checks whether this region has already written or read this variable, respectively, in which case the algorithm needs not check for conflicts or update read/write metadata. Otherwise, the analysis

Algorithm 1 REGION BOUNDARY [FastRCD-A]: thread T's region ends

```
1: incClock(T)                                  ▷Increment value returned by clock(T)
```

Algorithm 2 WRITE [FastRCD-A]: thread T writes x

```
1:  let c@t ← W_x
2:  if c@t ≠ epoch(T) then                      ▷First write to x by this region?
3:      if t ≠ T ∧ clock(t) = c then            ▷Write–write conflict?
4:          if deadlocked then
5:              throw consistency exception
6:          else
7:              Retry from line 1
8:      for all t' ↦ c' in R_x do
9:          if t' ≠ T ∧ clock(t') = c' then     ▷Read–write conflict?
10:             if deadlocked then
11:                 throw consistency exception
12:             else
13:                 Retry from line 1
14:     W_x ← epoch(T)                          ▷Update write metadata
15:     R_x ← ∅                                 ▷Clear read metadata
```

checks for write–write and then read–write conflicts (Algorithm 2) or write–read conflicts (Algorithm 3) by checking whether the last writer and reader regions are still ongoing (i.e., clock(t) = c). Note that checking for read–write conflicts involves checking for conflicts with every other thread's "last reader" region of x. If a conflict is detected, FastRCD-A tries to tolerate the region conflict by letting T wait (i.e., retry from line 1). After checking for conflicts, the analysis at a write or read updates the variable's write or read metadata, respectively. Additionally, the analysis at a write clears the read metadata. Instrumentation atomicity needs to be guaranteed at these operations, as discussed in Section 6.6.

Algorithm 3 READ [FastRCD-A]: thread T reads x

```
1: if clock(T) ≠ R_x[T] then                          ▷First read to x by this region?
2:     let c@t ← W_x
3:     if t ≠ T ∧ clock(t) = c then                   ▷Write–read conflict?
4:         if deadlocked then
5:             throw consistency exception
6:         else
7:             Retry from line 1
8:     R_x[T] ← clock(T)                              ▷Update read metadata
```

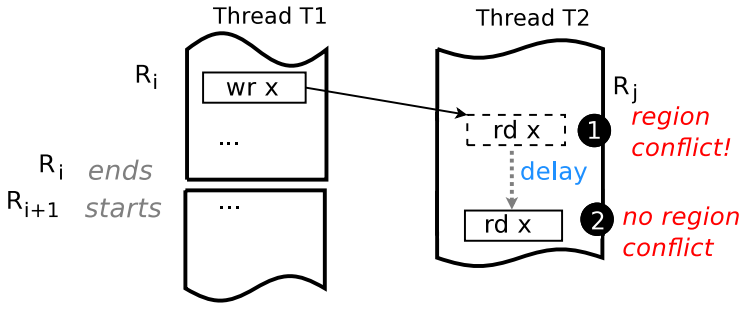


Figure 6.1: FastRCD-A avoids/tolerates some region conflicts: the read at (1) will cause a region conflict, but by waiting until (2), it succeeds without a consistency exception.

FastRCD differs from FastRCD-A in how it handles conflicts (lines 7 and 13 in Algorithm 2 and line 7 in Algorithm 3). Instead of checking for a deadlock, FastRCD simply throws a consistency exception.

Figure 6.1 shows an example of how FastRCD-A works. The boxes represent program regions in each thread; time flows downward. Suppose that thread T2 has not accessed x in its current region before reading x at time (1). When T2 tries to read x at (1), it causes a region conflict with the previous write to x by T1 in region R_i, because T1 is still executing R_i. T2 handles the region conflict by waiting until T1 finishes its region (R_i), at which point T2 retries its read at time (2) and continues execution safely.

Waiting-induced deadlocks. Due to waiting, FastRCD-A can run into *waiting-induced deadlocks*. FastRCD-A maintains a global *region wait-for graph* in order to detect these deadlocks. The graph's nodes represent regions labeled with an epoch c@t, and the graph contains at most one region per node. Each node has at most one wait-for edge from and to another region. When one thread needs to wait for another thread to finish, a wait-for edge is added into the wait-for graph. An edge-chasing deadlock detection algorithm is used to find cycles in the graph [83].

Precise exceptions. FastRCD-A (and FastRCD) provide *precise* consistency exceptions, meaning that it suspends a thread's execution immediately before it performs a conflicting access. Precise exceptions are easier to handle and debug than asynchronous exceptions [17].

6.2.2 Valor and Valor-A

As prior work and our evaluation show, FastRCD adds high runtime overhead, which is largely due to the cost of tracking last-reader metadata $\mathcal{R}_x$. Biswas et al. introduce an analysis called *Valor* that elides tracking of x's last-reader metadata [17]. Instead, Valor logs each read in a per-thread *read log*, and infers read–write conflicts lazily at region end.

In addition to the epoch(T) and clock(T) helper functions used by FastRCD, Valor uses the following notation (based closely on prior work [17]):

$\mathcal{W}_x$ – Represents last-writer metadata for variable x, as $\langle$v, c@t$\rangle$, where v is a *version* and c@t is an epoch. A variable's version starts at 0, and the analysis increments it at every write to the variable.

T.readLog – Represents thread T's read logs. Each entry in the log has the form $\langle x, v \rangle$, where x identifies the variable and v is x's version when it was read.

We extend Valor to *wait* at detected conflicts instead of throwing a consistency exception. We call the resulting analysis *Valor-A* (Available). Algorithms 4–6 show Valor-A's analysis at writes, reads, and region boundaries.

Algorithm 4 WRITE [Valor-A]: thread T writes variable x

```
1: let ⟨v, c@t⟩ ← W_x
2: if c@t ≠ epoch(T) then                          ▷First write to x by this region?
3:     if t ≠ T ∧ clock(t) = c then                 ▷Write–write conflict?
4:         if deadlocked then
5:             throw consistency exception
6:         else
7:             Retry from line 1
8:     W_x ← ⟨v+1, epoch(T)⟩                       ▷Update write metadata
```

Algorithm 5 READ [Valor-A]: thread T reads variable x

```
1: let ⟨v, c@t⟩ ← W_x
2: if t ≠ T ∧ clock(t) = c then                    ▷Write–read conflict?
3:     if deadlocked then
4:         throw consistency exception
5:     else
6:         Retry from line 1
7: T.readLog ← T.readLog ∪ {⟨x, v⟩}
```

Like FastRCD-A, Valor-A waits at detected write–write and write–read conflicts until the conflict no longer exists or the analysis detects a deadlock due to waiting on conflicts. However, unlike FastRCD-A, Valor-A cannot wait when it infers a read–write conflict: both the read and write accesses have already executed, so it is too late to try to avoid the

Algorithm 6 BOUNDARY [Valor-A]: T's region ends

```
1: for all ⟨x, v⟩ ∈ T.readLog do
2:     let ⟨v', c@t⟩ ← W_x
3:     if (v' ≠ v ∧ t ≠ T) ∨ v' ≥ v + 2 then
4:         Throw consistency exception                ▷Read–write conflict?
5: T.readLog ← ∅
```

conflict. Note that if the condition $(v' \neq v \wedge t \neq T) \vee v' \geq v + 2$ holds true, the condition cannot become false by waiting.

Valor-A (and Valor) therefore throw *asynchronous* consistency exceptions for read–write conflicts. Asynchronous exceptions can be problematic because the reader region has already executed using inconsistent values. This problem is particularly acute in the context of a memory- and type-unsafe language such as C++, where so-called "zombie" regions can lead to corruption that would not be possible in any RS execution [75, 47, 17]. Furthermore, asynchronous exceptions are difficult to use during development and debugging.

Thus, FastRCD is amenable to avoiding consistency exceptions, but it adds high run-time overhead. Valor improves performance by inferring read–write conflicts lazily, but it is not as well suited to avoiding consistency exceptions.

6.3 SIx: A New Strong Memory Model

The previous section presented our extensions to analyses that provide RSx, in an effort to avoid consistency exceptions as much as possible. Our evaluation shows that FastRCD-A and Valor-A generate significantly fewer exceptions than their non-waiting counterparts.

Can an approach provide availability comparable to or better than FastRCD-A's and performance comparable to Valor-A's? To achieve this goal, we *relax* consistency guarantees while still providing strong, principled semantics. This section introduces a new

memory consistency model called SIx, and the next section introduces an analysis for supporting SIx.

This new SIx memory model is based on providing *snapshot isolation* of regions, or *SI*. SI is weaker than region serializability (RS): under SI, a region that reads a variable can be concurrent with a region that later writes the variable. Although SI is weaker than RS, prior work (in database systems) has observed that behaviors that lead to SI instead of RS semantics are rare in practice [62, 95]. Our insight here is that *an approach that provides SI can potentially have lower runtime costs and fewer conflicts because it does not need to detect read–write conflicts accurately*.

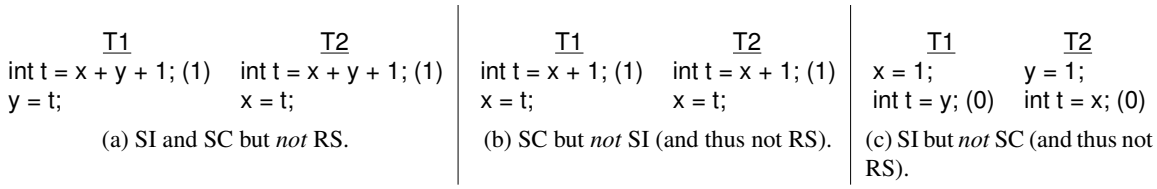
6.3.1 Snapshot Isolation of Regions

Prior work first introduced SI in the context of database processing [13]. Most of the database literature defines SI *operationally*. Here we give a definition based on execution equivalence, followed by a definition of *conflict SI*, which is a sufficient condition for SI. Both definitions are based closely on prior work.

Snapshot isolation. An execution is SI if it is equivalent to (has the same outcome as) some *serial* execution in which (1) each region R's reads[14] and writes *each* execute together as a group (without intervening accesses), (2) each region R's reads execute before its writes, and (3) another region Q's writes execute between R's reads and writes only if R and Q write distinct sets of variables [85].

In contrast, *serializability* demands equivalence to a serial execution in which all of a region's accesses (reads and writes) execute together without interruption [116, 114].

[14]These reads do *not* include reads from variables that the region has already written. These "local" reads always read the value that the region wrote.



(a) SI and SC but *not* RS.

(b) SC but *not* SI (and thus not RS).

(c) SI but *not* SC (and thus not RS).

Figure 6.2: Example executions comparing SI to RS and SC. Each thread executes just one region. Shared variables x and y are initially 0. Values in parentheses are the result of evaluating a statement's right-hand side.

Figure 6.2 shows a few examples to help understand the difference between RS and SI as well as sequential consistency (SC). Figure 6.2(a) shows by example that SI is weaker than RS. In contrast, Figure 6.2(b)'s execution violates SI (and thus RS): under SI, the regions cannot be concurrent because their write sets overlap. SI provides isolation, but not necessarily sequential consistency (SC), as Figure 6.2(c) shows. Despite not subsuming SC, SI is likely to be more intuitive (in addition to more practical to enforce) than SC: programmers already reason about code regions, and, while SI provides isolation of regions, SC subjects programmers to subtle reasoning about interleavings of memory accesses (Section 2.3).

Conflict SI. SI, as defined above, is not directly useful for designing approaches that provide SI. Instead, we use a slightly stronger property, *conflict SI*, a sufficient condition for SI that a dynamic analysis can check on the fly. Conflict SI is analogous to *conflict serializability* [155, 15].

The following definitions, which lead up to conflict SI, are closely based on prior work [8, 7]. Adya's dissertation proves that conflict SI is a sufficient condition for SI [7].

A multi-threaded execution consists of reads and writes executing in regions. The following notation describes read and write operations in an execution:

- $w_i(x_i)$: a write to variable x by region R_i
- $r_j(x_i)$: a read from x by region R_j, which sees the value written by region R_j ($i = j$ is allowed)

The following definition captures the notion of ordering in a multi-threaded execution:

Definition 1 (Time-precedes order)**.** *This order $\prec_t$ is a partial order over an execution's operations such that:*

1. *$s_i \prec_t e_i$, i.e., the start of a region precedes its end.*
2. *For any two regions R_i and R_j, either $e_i \prec_t s_j$ or $s_j \prec_t e_i$. That is, the end of one region is always ordered with start of every other region.*

Definition 2 (Conflict SI)**.** *An execution is conflict SI if the following conditions hold:*

1. *Two concurrent regions cannot modify the same variable. That is, for any two writes $w_i(x_i)$ and $w_j(x_j)$ such that $i \neq j$, either $e_i \prec_t s_j$ or $e_j \prec_t s_i$.*
2. *Every read sees the latest value written by preceding regions. That is, for every $r_i(x_j)$ such that $i \neq j$:*[15]
   (a) *$e_j \prec_t s_i$ and*
   (b) *for any $w_k(x_k)$ in the execution such that $j \neq k$, either $s_i \prec_t e_k$ or $e_k \prec_t s_j$.*

In the above definition, changing $s_i \prec_t e_k$ (in part 2b) to $e_i \prec_t s_k$ yields the definition of *conflict serializability*.

[15] if $i = j$, $r_i(x_j)$ sees the value from the latest $w_i(x_i)$.

6.3.2 A Memory Model Based on Snapshot Isolation

We introduce a new memory model called *SIx* based on snapshot isolation of regions (SI). Similar to RSx, under SIx, an execution either provides SI or throws a consistency exception—but only if the execution has a data race. In particular, SIx guarantees that each execution conforms to one of the following:

- a consistency exception, but only if the execution has a data race; or
- SI of regions if the execution does not throw a consistency exception.

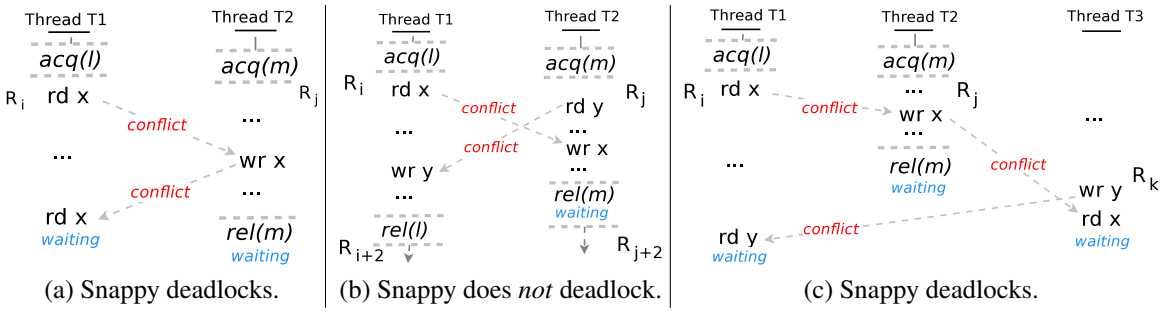
If an execution is data race free, SIx inherently ensures not only SI but also RS. SIx is strictly stronger than DRF0, and it relaxes semantics from RS to SI only for racy regions, which have undefined or weak semantics under DRF0.

6.4 Snappy: Runtime Support for SIx

This section introduces *Snappy*, a novel, software-only approach that provides the SIx memory consistency model. Like FastRCD-A and Valor-A, Snappy detects and waits at write–write and write–read conflicts. Snappy differs from FastRCD-A and Valor-A—and prior work in general—in two major ways. First, its support for SIx enables "tolerating" read–write conflicts by deferring waiting to the end of the writer region. Second, as a result of its handling of read–write conflicts, Snappy can detect read–write conflicts imprecisely, enabling less costly tracking of last reader(s), while still preserving SIx (no deadlock without a data race).

6.4.1 Tolerating Read–Write Conflicts

During a region's execution, Snappy tracks each shared variable's last writer and last reader(s) accurately (just like FastRCD-A). Last-writer tracking allows Snappy to detect



(a) Snappy deadlocks. (b) Snappy does *not* deadlock. (c) Snappy deadlocks.

Figure 6.3: Examples showing how Snappy works. Dashed lines indicate where Snappy increments its clock. The exact synchronization operations (e.g., acq(m) versus rel(l)) are arbitrary and not pertinent to the examples.

write–write and write–read conflicts precisely. Snappy handles read–write conflicts differently from both FastRCD-A and Valor-A: Snappy defers waiting on these conflicts until the end of the executing region, which still preserves SIx. With this design, Snappy opens new avenues for the improvement of availability and performance of exception-based memory models.

To present Snappy, we reuse clock(T), epoch(T), $\mathcal{R}_x$, and $\mathcal{W}_x$ from Section 6.2. In addition, we add the following notation specific to Snappy:

T.waitMap – Represents the regions that thread T is waiting on. T.waitMap is a map from a thread t to the latest clock value c of t that executed a read that conflicts with a write in T's current region. Clock values for threads not mapped in T.waitMap are considered to be 0.

Region boundaries. To support waiting at region boundaries, Snappy uses per-thread clocks to differentiate region execution from waiting at a region boundary. In particular, thread T's clock represents two execution states of T:

Algorithm 7 REGION BOUNDARY [Snappy]: T's region ends

```
1: incClock(T)                                   ▷Last region done; not ready to start next region
2: for each t ↦ c in T.waitMap do
3:     while clock(t) = c do                     ▷Read–write conflict still exists?
4:         deadlocked ← checkIfDeadlocked()
5:         if deadlocked then
6:             throw consistency exception
7: T.waitMap ← ∅
8: incClock(T)                                   ▷Ready to start next region
```

- clock(T) is *odd* if the region is executing. Note that initially clock(T) is 1.
- clock(T) is *even* if the region has finished executing but is waiting to tolerate read–write conflicts at a region boundary.

Algorithm 7 shows how Snappy maintains this invariant by incrementing clock(T) *both* before and after a region waits for tolerating any remaining read–write conflicts. While clock(T) is even, the algorithm checks whether read–write conflicts still exist; if the reader region is still executing (i.e., if clock(t) = c), the algorithm waits until the reader region finishes executing, throwing a consistency exception if Snappy detects deadlock.

Writes and reads. Algorithms 8 and 9 show the analysis that Snappy performs at each program write and read, respectively. The analysis differs from the analysis for FastRCD-A (Algorithms 2 and 3) in the following ways. First, in Snappy, a thread T waits at write–write and write–read conflicts (line 3 in Algorithm 8 and line 3 in Algorithm 9) for the writer region (executed by t) to finish any waiting at its region boundary. In order to take into account the two increments to clock(t) at a region boundary, T waits until the writer thread t's clock is at least two greater than the variable's clock c, i.e., clock(t) $\geq$ c + 2.

Algorithm 8 WRITE [Snappy]: thread T writes x

```
1:  let c@t ← W_x
2:  if c@t ≠ epoch(T) then                          ▷First write to x by this region?
3:      if t ≠ T ∧ clock(t) ≤ c + 1 then            ▷Write–write conflict?
4:          if deadlocked then
5:              throw consistency exception
6:          else
7:              Retry from line 1
8:      for all t' ↦ c' in R_x do
9:          if t' ≠ T ∧ clock(t') = c' then          ▷Read–write conflict?
10:             T.waitMap[t'] ← c'
11:     W_x ← epoch(T)                              ▷Update write metadata
12:     R_x ← ∅                                     ▷Clear read metadata
```

Algorithm 9 READ [Snappy]: thread T reads x

```
1:  if clock(T) ≠ R_x[T] then                       ▷First read to x by this region?
2:      let c@t ← W_x
3:      if t ≠ T ∧ clock(t) ≤ c + 1 then            ▷Write–read conflict?
4:          if deadlocked then
5:              throw consistency exception
6:          else
7:              Retry from line 1
8:      R_x[T] ← clock(T)                           ▷Update read metadata
```

Second, when T detects a read–write conflict (line 9 in Algorithm 8), instead of waiting, it records the conflicting thread t' and its current clock c' in T.waitMap. T.waitMap needs to maintain only the latest clock value for every other thread t' (multiple values are possible due to waiting on multiple threads), so the analysis updates T.waitMap only if the new value is greater than the old value.

Examples. Figure 6.3 shows examples of how Snappy enforces SIx. In the figure, concurrent regions access the shared variables x and y. The gray dashed lines along with the

synchronization operations, acq(l) and rel(l), indicate SFR boundaries. R_i and R_j are SFR identifiers, where i and j are per-thread clocks for the respective threads. In Figure 6.3(a), T2 waits on the read–write conflict at its region boundary, but T1 is unable to make progress due to a cyclic dependence. In Figure 6.3(b), a cyclic dependence exists, but each region can reach its region boundary, allowing each to proceed. In contrast, FastRCD-A deadlocks for this example. In Figure 6.3(c), Snappy deadlocks due to a cycle of transitive dependences involving two variables. Each thread gets stuck waiting: T1 and T3 at accesses, and T2 at a region boundary.

6.4.2 Detecting Read–Write Conflicts Imprecisely

Although tolerating read–write conflicts allows Snappy to avoid exceptions that FastRCD-A and Valor-A encounter, our evaluation shows that it incurs about the same (high) overhead as FastRCD-A in order to detect all conflicts accurately when they occur. In developing Valor, prior work shows that the main cost of FastRCD is in tracking the last reader(s) for each variable precisely to detect read–write conflicts accurately [17].

This section proposes an alternate design for Snappy that focuses on *the approximation of the precise last reader(s)* to improve performance. For clarity, the rest of the dissertation refers to the design of Snappy described in Section 6.4.1 as *Snappy-P* (Precise) and the alternate design introduced here as *Snappy-I* (Imprecise).

The key insight of having imprecise readers is to let a conflicting write conservatively wait on potential read–write conflict(s) if the write cannot infer accurate read–write conflict(s). Importantly, Snappy-I still provides SIx: although Snappy-I may wait on a false read–write conflict, any deadlock must include at least one (true) write–read or write–write

conflict (this fact holds for Snappy-P as well as Snappy-I), indicating a data race, as Section 6.5 shows.

In the design of *Snappy-P* from Section 6.4.1, each variable x has metadata both for writes ($\mathcal{W}_x$) and reads ($\mathcal{R}_x$). Both are needed for precise tracking of last reader(s). In particular, the read metadata needs to be inflated into a read map when there are multiple concurrent reader regions. Since Snappy-I does not require precise detection of read–write conflicts, it allows for a simpler and more efficient design.

Metadata representation. Like Snappy-P, Snappy-I maintains the epoch of the last writer for each shared variable x. However, it maintains precise last reader information only if there exists a single reader; for multiple readers, it does not maintain any information about them (so any ongoing region is a potential reader). As a result, Snappy-I represents a variable's last writer and reader metadata into a single unit of metadata (e.g., a single metadata word). This metadata has one of the following values:

WrEx$_{c@t}$: x was last accessed by region c@t, and that region performed a write to x.

RdEx$_{c@t}$: x was last accessed by region c@t, and that region performed only reads to x.

RdSh : At some point, there were multiple concurrent reader regions. Any ongoing region may have read x, but no ongoing region may have written x.

The first write in region c@t updates the variable's metadata to WrEx$_{c@t}$. Similarly, the first read in c (if there is no prior write in the same region) updates the metadata to RdEx$_{c@t}$. If a second read from a different thread reads the variable while the first read's region is still ongoing, Snappy-I promotes the metadata from RdEx$_{c@t}$ to RdSh. Since all states can

be encoded in a single metadata word, it is possible to use one *compare-and-swap* (CAS) instruction to update the metadata (Section 6.6).

Snappy-I's analysis. Algorithms 10 and 11 show the analysis that Snappy-I performs at each program write and read. In Algorithm 10, if the last write to x comes from the same region, the current write can skip the rest of the analysis operations (line 3) since the metadata does not need to be updated. If the last write is from the same thread T the write can update the metadata with the epoch of the current region (R_c). Otherwise, T handles $\mathsf{WrEx}_{*@t}$ and $\mathsf{RdEx}_{*@t}$ (* denotes "any clock value") as Snappy-P by detecting a write–write or a read–write conflict (lines 4–14). If the variable is in RdSh state, Snappy-I treats every other threads' ongoing region as having potential read–write conflicts with T's current write (lines 15–18).

In Algorithm 11, if the same region has already read or written the variable or the variable is in RdSh state, T does not need to update the metadata record (line 3). If the read is the first read in a region before any writes, the read overrides the metadata record from $\mathsf{WrEx}_{c'@t}$ to $\mathsf{RdEx}_{c@T}$, so that a write from a different thread can still detect a read–write conflict precisely at a $\mathsf{WrEx}_{c'@t}$ to $\mathsf{WrEx}_{c@T}$ transition instead of having a potential read–write conflict (an alternative is to change the metadata to RdSh, but it would lead to unnecessary imprecision and lower availability).

6.5 Correctness of Snappy

This section provides arguments that Snappy (in particular, Snappy-P) soundly and precisely provides SIx.

Algorithm 10 WRITE [Snappy-I]: thread T writes x

```
1:  let oldMetadata ← x.state
2:  let c ← clock(T)
3:  if oldMetadata ≠ WrEx_{c@T} then                    ▷First write to x by this region?
4:      if oldMetadata = WrEx_{⋆@t} then
5:          let c@t ← oldMetadata
6:          if clock(t) ≤ c + 1 then                    ▷Write–write conflict?
7:              if deadlocked then
8:                  throw consistency exception
9:              else
10:                 Retry from line 1
11:     else if oldMetadata = RdEx_{⋆@t} then
12:         let c@t ← oldMetadata
13:         if clock(t) = c then                        ▷Potential read–write conflict?
14:             T.waitMap[t] ← clock(t)
15:     else if oldMetadata = RdSh then
16:         for each thread t' do
17:             if t' ≠ T then                          ▷Potential read–write conflicts?
18:                 T.waitMap[t'] ← clock(t')
19:     x.state ← WrEx_{c@T}
```

Theorem 1. ***Snappy is sound:*** *If an execution completes without deadlock under Snappy, the execution conforms to snapshot isolation of synchronization-free regions (SI).*

We have not proved this theorem; instead we provide an argument for its correctness. Following from prior work, FastRCD and FastRCD-A ensure that executions that complete without exception or deadlock provide *conflict serializability*, a sufficient condition for region serializability (Section 6.3.1) [17]. The difference between FastRCD-A and Snappy is that Snappy allows execution to proceed past a read–write conflict until the region end, at which point execution can proceed if the reader region has finished executing, even if it is waiting on read–write conflicts.

Consider an execution that completes without deadlock. Let R_i be an ongoing region with a read to x, followed by a write to x by region R_j. Snappy detects the read–write

Algorithm 11 READ [Snappy-I]: thread T reads x

```
1:  let oldMetadata ← x.state
2:  let c ← clock(T)
3:  if oldMetadata ≠ RdEx_{c@T} ∧ oldMetadata ≠ WrEx_{c@T} ∧ oldMetadata ≠ RdSh then
4:      newReadMetadata ← RdEx_{c@T}
5:      if oldMetadata = WrEx_{*@t} then
6:          let c'@t ← oldMetadata
7:          if T ≠ t ∧ clock(t) ≤ c' + 1 then                ▷Write–read conflict?
8:              if deadlocked then
9:                  throw consistency exception
10:             else
11:                 Retry from line 1
12:     else if oldMetadata = RdEx_{*@t} then
13:         let c'@t ← oldMetadata
14:         if clock(t) = c' then                            ▷Concurrent reader?
15:             newReadMetadata ← RdSh
16:     x.state ← newReadMetadata
```

conflict and allows R_j to continue executing until region end, at which point R_j's thread waits until R_i reaches its region end. Thus, R_j cannot finish waiting at its region end until R_i ends.

According to the definition of conflict SI, $s_i \prec_t e_j$ due to the read–write conflict. Our soundness concern here is that the execution might also establish $e_j \prec_t s_i$ (which would violate conflict SI since $\prec_t$ is a partial order). However, if we suppose $e_j \prec_t s_i$, then there must exist a (potentially transitive) write–write or write–read conflict from R_j to R_i. Snappy in that case will ensure that R_i does not reach its end until R_j finishes waiting at its region end, which implies a deadlock due to the conclusion above about R_j waiting on R_i. A deadlock contradicts the original assumption that the execution completes without deadlock. We thus conclude that Snappy's relaxation for read–write conflicts does not lead to violations of SI.

Lemma 1. *If an execution deadlocks under Snappy, it has a write–write or write–read conflict.*[16]

Proof. Suppose an execution deadlocks under Snappy, but the execution has no write–write or write–read conflicts. Since Snappy waits at program writes and reads only if there is a write–write or write–read conflict (Algorithms 8 and 9), Snappy does not wait at program reads or writes for this execution.

The execution thus deadlocks by waiting at Snappy's only other waiting point, a region boundary (lines 3 in Algorithm 7). Let T be a thread that is waiting at a region boundary as part of a deadlock. According to the wait condition, clock(t) $=$ c. The value c comes from a t $\mapsto$ c entry in T.waitMap (line 10 in Algorithm 8), which in turn comes from a t $\mapsto$ c entry from $\mathcal{R}_x$ (line 8 in Algorithm 9). $\mathcal{R}_x$[t] comes from clock(t), which must be odd because t is executing a region. So c must be odd in the wait condition at T's region boundary.

Since clock(t) $=$ c, clock(t) is also odd, implying that thread t is in the middle of executing a region (i.e., not at a region boundary). Since Snappy waits only at region boundaries for this execution, t's region eventually finishes, incrementing clock(t) and thus negating thread T's waiting condition, i.e., $\neg$(clock(t) $=$ c). This contradicts the earlier conclusion that T is waiting as part of a deadlock. □

Theorem 2. ***Snappy is precise:*** *If an execution deadlocks under Snappy, it has a region conflict and a data race.*

Proof. Suppose an execution deadlocks under Snappy. By Lemma 1, the execution has a region conflict, which is a sufficient condition for a data race. □

[16]Section **??** defines a region conflict.

6.6 Implementation

All implementations instrument the same points (field and array element accesses), demarcate regions in the same way (at lock, monitor, thread, and volatile operations), and reuse code as much as possible. The compilers instrument all application code, and the application calls an instrumented compiled version of the Java libraries (the JVM, which is written in Java, calls a separately compiled, uninstrumented version of the libraries).

FastRCD-A and Snappy-P add two words of metadata for tracking writes and reads. While each variable has only a write epoch, its read metadata can be inflated to a pointer that points to a read map. In contrast, Snappy-I use one word of metadata: 21 bits for the clock, 9 bits for the thread ID, and 2 bits for encoding the state (write-exclusive vs. read-exclusive vs. read-shared). Snappy-I can reset clocks to either 0 or 1 at full-heap garbage collection to avoid overflow [17]; or it can ignore overflow, in which case false positive conflicts due to wraparound are unlikely. In all implementations, each per-field metadata is laid out beside the field, while an array's per-element metadata is referenced indirectly through the array's header.

Instrumentation atomicity. Following FastRCD, FastRCD-A, and Snappy-P guarantee instrumentation atomicity by "locking" one of the variable's metadata words (using an atomic operation and spin loop) and "unlocking" (using a store and fence) when updating the metadata. The instrumentation does not perform any synchronization when the instrumentation performs no metadata updates (for a read or write in the same region). These analyses require lock and unlock operations to ensure instrumentation atomicity, since metadata is two words or more (due to a possible read map). In contrast, following Valor, Valor-A and Snappy-I use a single word of metadata per variable, and ensure

instrumentation atomicity by using a single atomic operation at the end of the analysis to atomically update the state.

Waiting at conflicts. Instrumentation for the FastRCD-A, Valor-A, Snappy-P and Snappy-I waits when it detects a region conflict. In order for a thread T to wait on another thread t's clock to change, T waits on a monitor associated with t; t notifies T that it has finished its region by broadcasting on the monitor at region boundaries.

6.7 Evaluation

This section evaluates the availability, performance, scalability, space usage, and other characteristics of our approaches and existing approaches for providing RSx and SIx.

6.7.1 Runtime characteristics.

Table 6.1 shows characteristics of the evaluated programs. The *Threads* columns report both threads created and maximum threads active at any time. The remaining columns show statistics collected with Snappy-P (the statistics from other configurations are similar, since these statistics are not specific to configurations). The *Memory accesses* are executed memory accesses (loads and stores of fields and array elements), which all configurations instrument. The *Conflicts* column shows how many conflicts of each type occur (during execution under Snappy-P). Conflicts vary significantly in count and type across programs, but they are generally many orders of magnitude smaller than total memory accesses, except for avrora9, which incurs millions of write–read conflicts.

The last two columns report executed synchronization-free regions (SFRs) and average memory accesses executed in each SFR. All programs except sunflow9 perform synchronization at least every 720 memory accesses.

	Threads		Memory accesses		Conflicts			Dyn.	Avg. accesses
	Total	Max live	Reads	Writes	Write–write	Write–read	Read–write	SFRs	per SFR
eclipse6	18	12	4,500M	1,400M	0	3.2K	21	150M	40
hsqldb6	402	102	250M	31M	0	33	1.9	11M	25
lusearch6	65	65	1,100M	400M	0	96	0	9.9M	150
xalan6	9	9	990M	220M	1.4K	520	26	58M	21
avrora9	27	27	900M	440M	380K	3.4M	25K	3.9M	350
jython9	3	3	720M	230M	0	0	0	100M	9.2
luindex9	2	2	290M	97M	0	0	0	540K	720
lusearch9*	32	32	1,100M	350M	38	5.7K	22	7.2M	210
pmd9	5	5	290M	97M	6.6K	5.3K	120	2.4M	160
sunflow9*	64	32	6,700M	720M	0	3.9	4.9	16K	450K
xalan9*	32	32	940M	210M	200	1.4K	42	22M	53

Table 6.1: Runtime characteristics of the evaluated programs, rounded to two significant figures. *Three programs support varying the number of active application threads; by default, this value is equal to the number of cores (32 in our experiments).

6.7.2 Availability

Under the DRF0 consistency model, data races are essentially errors since they have undefined or weak semantics (Section 2.3). However, in practice, language and hardware implementations typically ignore data races silently (rather than treating them as fail-stop errors), and thus real programs—including those we evaluate—often contain data races that commonly manifest but do not cause problems under typical compilation and execution environments. In contrast, by providing RSx and SIx, our work follows a line of research that treats some or all data races as errors [22, 98, 101, 137, 17, 58, 156, 38]. In order to avoid frequent exceptions in production environments, developers would need to identify and fix data races that commonly lead to consistency exceptions when using Valor or Snappy or another technique that provides RSx or SIx (e.g., during in-house, alpha, and beta testing).

Since the programs we evaluate have *not* been developed or debugged under the assumption that data races are (fail-stop) errors, they throw consistency exceptions frequently under RSx and SIx. We compare the numbers of consistency exceptions generated under

different approaches for RSx and SIx. From that, we extrapolate which approaches would be more likely to avoid exceptions *if programs were developed and debugged to avoid consistency exceptions*.

Table 6.2 compares consistency exceptions raised by FastRCD, Valor, FastRCD-A, Valor-A, Snappy-P, and Snappy-I. Our implementations do not actually generate exceptions; rather, they simply report the exception and allow execution to proceed. In case of a deadlock, the thread that created the cycle of dependences is allowed to proceed immediately. FastRCD and Valor report an exception whenever they detect a conflict. FastRCD-A, Valor-A, Snappy-P, and Snappy-I report an exception whenever they detect a deadlock. Valor and Valor-A also report an exception whenever they infer a conflict via a read validation failure.

Since some executed regions may have many related conflicts, the first row for each program is the number of dynamic regions that report at least one exception. The second row is the number of consistency exceptions reported during the program execution. Each result is the mean of exceptional regions or exceptions, $\pm$ 95% a confidence interval.

Waiting at conflicts. We first consider the effect of *waiting* at conflicts rather than generating an exception, i.e., Section 6.2's innovation that yields FastRCD-A and Valor-A. Based on comparing FastRCD-A with FastRCD and Valor-A with Valor, the effect of waiting at conflicts is substantial, reducing regions that report an exception by up to three orders of magnitude. Our evaluation of waiting at conflicts—which to our knowledge is the first such evaluation—suggests that this approach is generally effective at increasing the ability to avoid consistency exceptions while still preserving consistency models such as RSx and SIx.

	FastRCD	Valor	FastRCD-A	Valor-A	Snappy-P	Snappy-I
eclipse6	**6.8K** ± 10K	**260** ± 510	**0.2** ± 0.5	**0.2** ± 0.3	**0.3** ± 0.6	**1.2** ± 1.6
	(**14K** ± 24K)	(**1.8K** ± 3.5K)	(**0.2** ± 0.5)	(**0.7** ± 0.9)	(**0.3** ± 0.6)	(**1.2** ± 1.6)
hsqldb6	**27** ± 2.7	**73** ± 2.6	**0.8** ± 0.5	**0.6** ± 0.3	**0.1** ± 0.1	**1.0** ± 0.5
	(**53** ± 5.6)	(**140** ± 5.4)	(**0.8** ± 0.5)	(**0.6** ± 0.3)	(**0.1** ± 0.1)	(**1.2** ± 0.6)
lusearch6	**0.1** ± 0.1	**0.2** ± 0.2	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0
	(**0.1** ± 0.1)	(**0.2** ± 0.2)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)
xalan6	**48** ± 1.7	**53** ± 1.7	**12** ± 1.3	**25** ± 1.1	**5.5** ± 1.4	**10** ± 1.3
	(**120** ± 5.8)	(**93** ± 3.4)	(**12** ± 1.3)	(**39** ± 1.1)	(**5.5** ± 1.5)	(**11** ± 1.4)
avrora9	**200K** ± 3.1K	**230K** ± 2.6K	**38K** ± 760	**28K** ± 400	**16K** ± 430	**18K** ± 430
	(**610K** ± 6.4K)	(**670K** ± 9.7K)	(**39K** ± 820)	(**36K** ± 920)	(**24K** ± 730)	(**27K** ± 760)
jython9	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0
	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)
luindex9	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0
	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)
lusearch9	**63** ± 9.7	**62** ± 7.3	**62** ± 7.6	**13** ± 3.3	**20** ± 3.7	**19** ± 5.7
	(**110** ± 25)	(**93** ± 13)	(**62** ± 7.6)	(**13** ± 3.3)	(**21** ± 4.5)	(**21** ± 6.6)
pmd9	**450** ± 150	**370** ± 150	**91** ± 10	**71** ± 7.2	**52** ± 9.5	**77** ± 7.2
	(**3.3K** ± 520)	(**3.5K** ± 930)	(**93** ± 11)	(**170** ± 62)	(**110** ± 51)	(**120** ± 17)
sunflow9	**6.1** ± 1.8	**11** ± 3.6	**0** ± 0	**0** ± 0	**0** ± 0	**4.6** ± 1.1
	(**23** ± 6.3)	(**37** ± 14)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**4.7** ± 1.2)
xalan9	**330** ± 35	**3.0K** ± 490	**4.7** ± 3.7	**40** ± 3.6	**0.3** ± 0.4	**16** ± 3.2
	(**1.5K** ± 220)	(**6.2K** ± 1.2K)	(**4.7** ± 3.7)	(**40** ± 3.6)	(**0.3** ± 0.4)	(**19** ± 4.5)

Table 6.2: The number of consistency exceptions reported by approaches that provide RSx and SIx. For each program, the first row is dynamic regions that report at least one exception, and the second row is dynamic exceptions reported. Reported values are the mean of 10 trials, including 95% confidence intervals, rounded to two significant figures (if $\geq$1.0).

RSx versus SIx. A key intended benefit of Snappy is that by providing SIx, it can potentially avoid consistency exceptions encountered by providing RSx. To evaluate this benefit, we compare primarily exceptions generated by Snappy-P versus FastRCD-A, which are identical except that Snappy-P can tolerate read–write conflicts. We find that Snappy-P generally avoids consistency exceptions compared to FastRCD-A: Snappy-P has fewer exceptional regions than FastRCD-A for 10 programs, never has more exceptional regions than FastRCD-A, and for the other 1 program (i.e., eclipse6) there is no statistically significant difference.

Relaxing Snappy's precision. Another potential benefit of Snappy is that by relaxing precision, it can improve performance. In order to evaluate the *cost* of this, we compare reported consistency exceptions for Snappy-P and Snappy-I. Unsurprisingly, Snappy-I reports more exceptions than Snappy-P, since Snappy-I introduces waiting at region boundaries for spurious read–write conflicts. This effect is mixed across programs: for 6 programs Snappy-I generates more exceptions than Snappy-P, whereas we find no statistically significant difference for the other 5 programs.

Overall, waiting at conflicts and providing SIx instead of RSx both increase availability, while relaxing Snappy's precision decreases availability, but perhaps not excessively. The suitability of each approach not just on its availability but also on its performance, discussed next.

6.7.3 Performance

This section measures and compares the effect on performance of various approaches for providing strong memory models. Since executions are multi-threaded, performance overheads include not just instrumentation overhead but also time spent on waiting, which differs among approaches due to different conditions for waiting. Section 6.7.5 attempts to separate out this cost by measuring performance for varied application thread counts.

Figure 6.4 shows the runtime overhead added by the configurations from Section 6.7.1 to execution on an unmodified JVM. As prior work shows, FastRCD adds high runtime overhead in order to maintain last readers, while Valor incurs significantly lower overhead by logging reads locally and validating them lazily [17]. FastRCD-A and Valor-A each add additional overhead over FastRCD-A and Valor-A, respectively, due to the time spent waiting at conflicts.

Snappy-P tracks write and read metadata in the same way as FastRCD-A, so it unsurprisingly incurs similar overhead. However, Snappy-P incurs slightly less overhead than FastRCD-A because Snappy-P provides SIx and thus can relax its waiting at read–write conflicts (by deferring waiting until region end; Section 6.4). However, Snappy-P adds 260% overhead on average in order to provide precise conflict detection. In contrast, Snappy-I enables a significantly faster analysis that slows programs by 128% on average.

To understand the performance difference between Snappy-I and Snappy-P, we implemented a configuration (not shown in the figure) that tracks both read and write metadata in a way similar to Snappy-P (i.e., separate metadata words for the writer and reader(s), except that it does not track multiple readers precisely, instead using a simple "read shared" when multiple concurrent reader regions exist. On average this configuration incurs 75% of the overhead that Snappy-P incurs over Snappy-I, suggesting that most but not all of Snappy-I's performance advantage comes from its ability to represent its metadata in a single metadata word, leading to significantly simpler and cheaper instrumentation.

To isolate Snappy-I's overhead due to waiting at conflicts (versus instrumentation overhead), we also evaluated a Snappy-I configuration (not shown in the figure) that does not wait at conficts, but instead allows a thread to proceed immediately after detecting a region conflict. Compared with this "no waiting" configuration, the default Snappy-I configuration adds only 9.5% additional overhead (relative to baseline, unmodified execution), suggesting that little of Snappy-I's overhead is due to waiting at conflicts. This result makes sense conflicts are usually many orders of magnitude smaller than total memory accesses. The entire execution time therefore is still dominated by instrumentation overhead added to memory accesses.

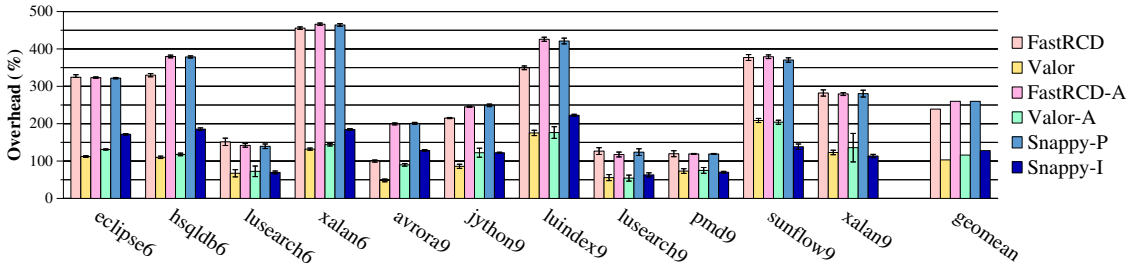


Figure 6.4: Runtime overhead added to unmodified Jikes RVM by FastRCD, Valor, and our implementations of FastRCD-A, Valor-A, Snappy-P, and Snappy-I.

6.7.4 Performance–Availability Tradeoff

In order to evaluate availability and performance together, we plot the previous availability and performance results in Figure 6.5. The x-axis is the geomean of runtime overhead across all programs. The y-axis is availability, which is defined as the geomean of memory accesses performed without interruption by a consistency exception. That is, for each program, $\textit{availability} = \frac{\#\ \textit{memory accesses}}{\#\ \textit{consistency exceptions}+1}$. (In contrast, taking the geomean of exceptions would be problematic because some values are 0.) Values closer to the top left corner represent a better performance–availability tradeoff.

Valor has the best performance, but its availability is significantly worse than FastRCD-A, Valor-A, Snappy-P, and Snappy-I. Snappy-P has the best availability, but its performance overhead is relatively high. Snappy-I and Valor-A arguably have the best tradeoff between availability and performance. We note that Snappy-I has lower space overhead than other implementations (Section 6.7.6), and it does not have Valor-A's disadvantages of asynchronous exceptions and safety issues for unsafe languages (Section 6.2.2).

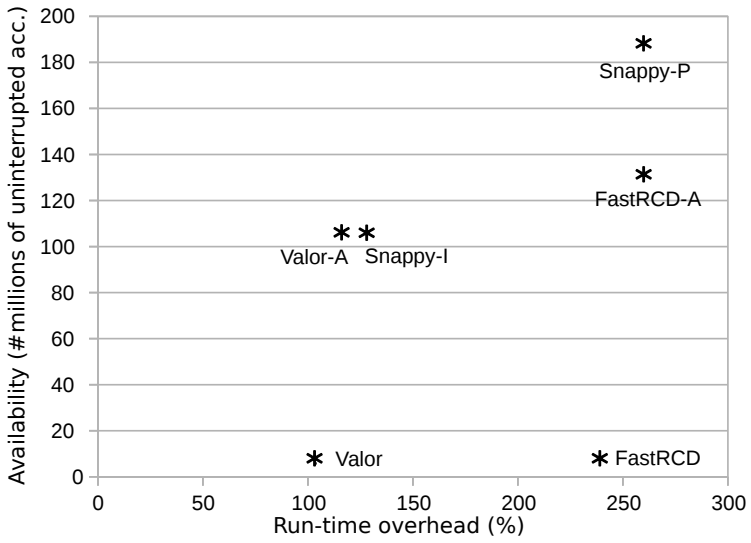
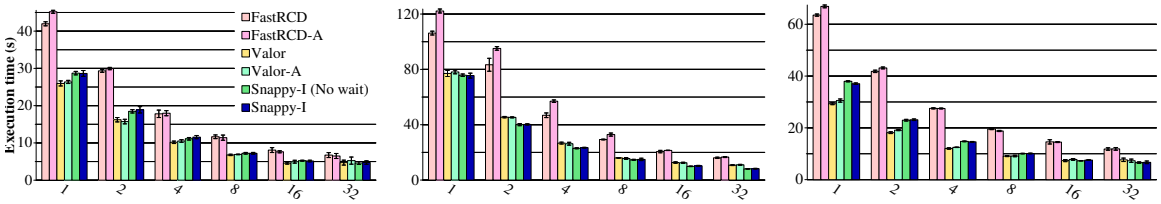


Figure 6.5: The comparison of performance and availability of different memory models.

6.7.5 Scalability

Approaches that avoid exceptions by waiting at conflicts—FastRCD-A, Valor-A, Snappy-P, and Snappy-I—incur not only instrumentation overhead but also overhead due to waiting. In an effort to separate out the conflicts, this section evaluates scalability across varying numbers of applications threads. Three of the evaluated programs support varying the number of application threads (Table 6.1). Figure 6.6 compares, for 1–32 application threads, the execution time of the waiting and non-waiting versions of implementations that are otherwise identical. We leave out Snappy-P and only show Snappy-I since the measurement here is whether waiting at conflicts hurt scalability instead of instrumentation overhead. We use Snappy-I (No wait) (introduced in the last subsection) as a comparison configuration to Snappy-I.



(a) lusearch9 (b) sunflow9 (c) xalan9

Figure 6.6: Execution time of the configurations that can incur waiting versus configurations that do not incur waiting, for 1–32 application threads. The legend applies to all graphs.

Overall, supporting RSx and SIx with waiting has no detrimental effect on scalability. For all three benchmarks, the waiting versions (FastRCD-A, Valor-A, Snappy-I) scale equally well as the non-waiting configurations (FastRCD, Valor, Snappy-I (No wait)).

6.7.6 Space Overhead

The approaches incur space overhead in order to represent write and read metadata. Figure 6.7 shows the space overhead of all configurations that wait at conflicts, relative to unmodified JVM execution. For each execution, we define its space usage as the maximum memory used after any full-heap garbage collection (GC). We omit luindex9 since its baseline execution triggers no full-heap GCs.

On average, FastRCD-A adds 105% space overhead in order to maintain precise write and read metadata, which is particularly costly for variables with concurrent reader regions. Snappy-P adds similar memory overhead (103% on average), which makes sense because it maintains the same metadata as FastRCD-A. Although Valor-A avoids storing per-variable read metadata, it still adds high space overhead (110% on average) due to maintaining per-thread read logs. Per-program space overheads depend largely on thread counts and region size (Table 6.1).

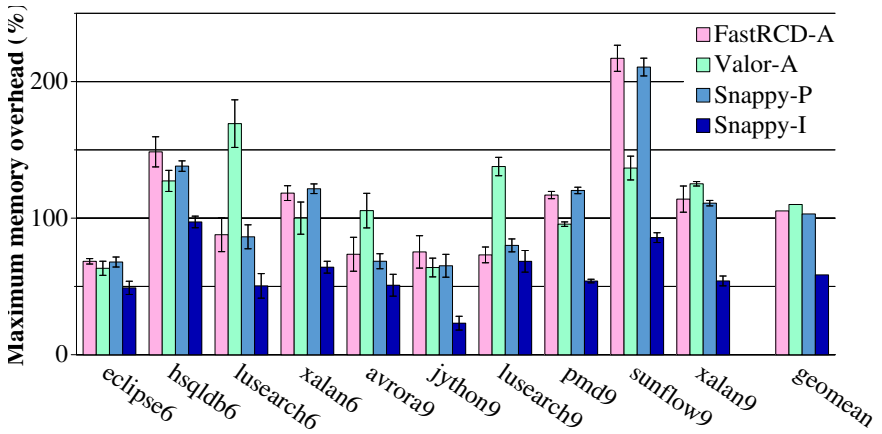


Figure 6.7: Runtime space overhead of FastRCD-A, Valor-A, Snappy-P, and Snappy-I.

Snappy-I adds 58% average space overhead, about half as much as the other approaches. Snappy-I uses less space by not maintaining precise information about reads—particularly in the case of multiple readers—whereas the other configurations maintain precise information about reads (including Valor-A, which maintains them in pre-thread logs). Furthermore, Snappy-I is able to use a single metadata word per field and array element for write and read metadata.

6.8 Conclusion and Interpretation

Memory models are the foundation for reasoning the behavior of concurrent programs. Existing approaches providing strong memory model RSx suffers from availability issues which limit its practicality. In this chapter, we mediated the need of high availability and simplicity for strong memory models for the purpose of practicality and programming ease. We considered designs and optimizations to existing approaches that provide RSx, which our evaluation showed significantly improved availability. This led to an alternative strong and intuitive memory model SIx that enabled a higher degree of performance and simplicity

by relaxing some of the constraints imposed by RSx. We then introduced a novel approach for enforcing SIx called Snappy. The rest of the chapter was devoted to a thorough analysis of these two memory models with the hope of clarifying and resolving the various trade-offs. Our evaluation showed that Snappy-P reduced the chances of unavoidable consistency exceptions under SIx significantly. A high-performance version of Snappy-I achieved much lower overhead, but in turn lost the availability benefit provided by SIx over RSx.

Overall, this work has been shed light on the complex trade-offs that arise in choosing an appropriate strong memory consistency model, which helps enhance programmability. It demonstrates efficient and scalable techniques for implementing a strong memory model with high availability and performs detailed quantitative analysis of the performance benefits of various models.

Chapter 7: Related Work

In this chapter, we describe work that is related to the dissertation's contributions and compares them.

7.1 Tracking Dependences

Biased Locking. Prior work shows how to acquire locks without performing atomic operations in the common, non-conflicting case. In *biased locking*, a lock is "owned" by one thread, which can acquire the lock without using synchronization [82, 32, 123]. Before another thread can acquire the lock, it must "coordinate" with the owning thread—a relatively expensive operation—to ensure the owning thread safely releases the lock.

Hindman and Grossman use biased per-object locks in a strongly atomic STM to reduce synchronization costs [80]. However, biased locks do not work well for objects that are mostly read shared (i.e., multiple readers, no writers), since interleaved reads by different threads lead to expensive coordination between threads. Some prior work provides biased reader–writer locks [28, 126, 129, 149], although distributed shared-memory work targets software cache coherence instead of detecting dependences [126, 129], and von Praun and Gross's model requires an atomic operation at each read-shared access [149]. However, but to our knowledge, no prior work applies these ideas to build runtime support such as

STM or record & replay systems. Furthermore, in Hindman and Grossman's STM, a conflicting access triggers conflict detection and resolution as part of coordination; however, a conflicting access always aborts the *other* thread's transaction. Our LarkTM introduced in Chapter 4 supports aborting either thread, enabling flexible contention management.

Commodity hardware support for capturing cross-thread dependences. Recently, Intel's Haswell architecture provides *restricted transactional memory* (RTM), best-effort HTM support with an upper bound on shared-memory accesses in a transaction [121]. It might seem at first that RTM could achieve the same goals as reader–writer locks, but recent work suggests otherwise: RTM struggles to outperform locking based on atomic operations, let alone biased locking, which tries to avoid atomic operations. Recent work uses RTM to replace per-object locks for detecting data races, but several critical sections need to be combined to overcome the costs of RTM [103]. Similarly, recent work finds that an RTM transaction needs to be expanded to replace at least 3–4 atomic operations, in order to amortize the overhead of a transaction [122, 159]. In contrast, biased reader–writer locks try to avoid atomic operations, and can provide substantially lower overhead than traditional locks or RTM.

7.2 Transactional Memory

McRT-STM and Bartok. Both McRT-STM and Bartok support weak atomicity and use pessimistic concurrency control writes and optimistic concurrency control for reads. In particular, a transaction has to acquire locks to perform updates and resolve writes/writes conflicts eagerly. For reads, a transaction maintains read logs and performs read validation to check reads/writes conflicts at the end of a transaction.

TL2. Both McRT-STM and Bartok still need encounter-time locking for accessing transactional writes. TL2 provides commit-time locking which can avoid using locks when a transaction is executing. A commit-time locking mechanism also requires the transaction to exploit lazy version management to make sure all transactional reads can read consistent value. Though TL2 avoids the synchronization on writes, it needs a global clock to assign a unique version number at the end of each transaction. Though the contention on the global lock for every transaction might not affect the performance too much when transactions are long, it might become a performance worry if extending TL2 to support strong atomicity as each access need to be treated as a unary transaction.

InvalSTM and DSTM. InvalSTM detects conflicts with other transactions when committing writes lazily, potentially aborting either transaction [69]. A few STMs, like LarkTM, have detected read and write conflicts eagerly and supported aborting either transaction (e.g., DSTM [78]), but fully eager conflict detection has added very high overhead.

Dynamic-NAIT. Schneider et al. and Bronson et al. reduce strong atomicity's cost for languages with dynamic class loading by optimistically performing static analysis incrementally [128, 31]. Whenever newly loaded code invalidates an assumption about a non-transactional access being unable to conflict with transactions, the dynamic compiler recompiles the non-transactional access's method. In the best case, this approach adds no overhead for non-transactional accesses. In the worst case, numerous mis-speculations lead to substantial recompilation, and imprecise static analysis leads to unnecessary instrumentation of non-transactional accesses. This approach is complementary to the inexpensive instrumentation LarkTM adds at non-transactional accesses.

Hardware-memory-protection-based STM. Abadi et al. present a strongly atomic STM that uses commodity-hardware-based memory protection to detect non-transactional conflicting accesses [3]. Transactions and non-transactional accesses use separate page mappings, enabling triggering of traps only when non-transactional code accesses physical pages being accessed concurrently by transactions. Two significant costs are (1) frequently changing page permissions in transactions and (2) false conflicts due to page granularity. The design ameliorates these costs through various static and dynamic techniques. It uses static analysis to detect accesses to objects that can only be accessed inside or outside transactions, similar to other prior work [128, 135, 31]. It reduces false conflicts (as well as expensive true conflicts) on the fly by recompiling some non-transactional accesses to use transactional instrumentation rather than relying on page protection, similar to prior dynamic work [128, 31]. These optimizations are complex and may not work well for all programs, although they work well for the implementation and programs evaluated by Abadi et al. [3], adding 25% overhead on average to an existing weakly atomic STM [76]. This approach is complementary to our work, which provides cheap non-transactional and transactional barriers, and could provide a cheaper fallback when recompiling non-transactional accesses.

Hardware transactional memory. Transactional memory can be supported either via hardware or software. Hardware TM (HTM) detects and resolves conflicts efficiently by piggybacking on cache coherence protocols; provides versioning efficiently by extending caches; and provides strong atomicity naturally (e.g., [79, 12, 107]). However, hardware manufacturers have moved slowly on HTM support, which requires changes to already-complex cache coherence protocols, cache architectures, and memory subsystems. Sun's

Rock processor included HTM support and the silicon was built, but it was never sold commercially [53]. Azul Systems produces custom processors with HTM support that is tightly integrated with Azul's custom Java virtual machine [44]. More recently, Intel's Haswell architecture, AMD's Advanced Synchronization Facility, and IBM's Blue Gene/Q provide "best-effort" HTM support with an upper bound on shared-memory accesses in a transaction [43, 153, 121]. None of these limited HTMs can provide language-level TM since they cannot guarantee a transaction will ever succeed. They require a software fallback, and efficient STM is critical for providing efficient hybrid TM [12].

7.3 Memory Models

Previous chapter covered RSx-based approaches, and this section covers other related work.

Enforcing region serializability. Prior work has *enforced* serializability of synchronization-free regions (RS), relying on heavyweight support for speculative execution [114]. The costs and complexity for enforcing RS are similar to those encountered in software and (unbounded) hardware transactional memory implementations (e.g., [72, 11, 49, 55, 160]). Furthermore, operations such as I/O and system calls are not generally amenable to speculative execution [75].

Serializability of bounded regions. Prior work supports a memory model based on serializability of regions *smaller* than SFRs [101, 137, 9, 99, 131]. This approach can enable architecture or analysis support that is less complex than for full SFRs. In addition to being weaker than RSx, bounded region serializability requires restricting compiler optimizations across region boundaries. Our SIx model and Snappy analyses relax RSx in a

different way: they retain full SFRs but provide isolation but not atomicity, in an effort to improve availability and reduce costs and complexity.

Detecting and tolerating data races. Data race detectors that soundly and precisely check the happens-before relation [87] can provide RSx, by throwing a consistency exception on every detected data race. However, state-of-the-art happens-before detectors slow programs by nearly an order of magnitude on average or rely on custom hardware support [63, 156, 58]. (Although prior work reports slowdowns of only 2X for *Goldilocks* [58], Flanagan and Freund show that using realistic methodology would incur an estimated 25X average slowdown [63].

Recent work introduces a data race detector called *Clean* that detects write–write and write–read races but not read–write races [130]. By providing fail-stop semantics at the detected data races, Clean eliminates some of the most egregious weak memory model behaviors (e.g., so-called "out-of-thin-air" violations [100, 25]), Clean and Snappy both relax the requirement of detecting read–write conflicts precisely. However, Clean incurs high overhead in order to track the happens-before relation. It inherently cannot tolerate detected data races: waiting at an access can avoid a detected conflict but not a detected data race. Although Clean can avoid some erroneous behaviors, it does not provide SIx or any strong guarantee of isolation of regions.

Deterministic execution. Systems that provide deterministic multi-threaded execution have employed mechanisms that are related to those used by FastRCD-A, Valor-A, and Snappy. *DMP* delays each region's writes until a point where all regions perform writes at the same time, in order to produce a deterministic outcome [52, 14]. *Dthreads* exploits existing relaxed memory models in order to perform loads and stores in isolation and merge

them at synchronization operations [96]. In contrast, Snappy detects conflicts in order to provide the SIx memory model, and it waits at conflicts in an effort to increase availability.

Snapshot isolation in other contexts. Database management systems routinely support SI instead of strict serializability semantics [59, 62, 118]. These systems typically implement SI using *multi-versioning* to track multiple versions of data, based on a globally ordered timestamp that provides a total order for all committed transactions. Maintaining globally ordered transactions and multiple versions of data at the programming language level would likely incur high overhead and poor scalability.

Prior work has used SI as the isolation model for software transactional memory (STM) systems [85, 94]. In contrast, our work focuses on SI-based semantics for memory consistency models. Furthermore, the database and STM work executes transactions speculatively (i.e., conflicts lead to rollbacks), while our work converts data races with ill-defined semantics to well-defined behaviors and employs SI instead of RS in an effort to increase availability and reduce costs and complexity.

Chapter 8: Conclusion

This section concludes with a summary of the work presented in this dissertation and a discussion of future work.

8.1 Summary and Impact

In recent years, CPU clock and speeds have barely increased, but multicore computers have become common. As the trend of increasing cores on a single machine continues, programmers need to develop concurrent programs in order to make the most use of modern multicore parallel hardware. In order to have explosive growth of adopting concurrent programming, programming languages and tool support have to become more mature and practical.

We argue that using novel concurrency control techniques and mechanisms in runtime support can make memory dependence tracking, concurrent programming models, and memory consistency models substantially more efficient, powerful, and scalable.

Our dependence tracking approach empowers developers to build efficient runtime support for concurrent programs. The relaxed tracking technique hides coordination latency incurred at tracking cross-thread dependences, and we demonstrate with client analyses that this technique can be used to build more efficient runtime support, which will improve programmers' productivity, debuggability, and ability to analyze concurrent programs.

We show that STM can be practical by designing and building an STM system that has significantly lower overhead than prior work. Not just that, our STM is also scalable while at the same time provides strong semantics, which helps programmers to reason about program behaviors in a clearer way and makes programs more reliable. It further provides strong progress guarantees, which makes it well suit for building hybrid TMs considering that existing HTMs do not provide any progress guarantees.

Finally, we dig into today's incomplete memory models that give clear behavioral guarantees only to data-race-free program but can lead to not well defined semantics for racy executions. Recent work has introduced the RSx memory model, which provides strong guarantees to all executions rather than just data-race-free executions. However, there are also two significant challenges in bringing RSx into practice: availability and performance. We address these challenges by introducing a way to tolerate consistency exceptions with best-effort, which improves availability. We then introduce a new memory model SIx and a novel approach for enforcing SIx during runtime. We show that our approach reduces the chances of unavoidable exceptions significantly. Overall, our work represents an advance in the state of the art by introducing novel memory models and runtime approaches that enable new and compelling points in the performance–availability space for strong memory consistency models that use fail-stop semantics.

By addressing challenges in memory dependence tracking, concurrent programming models, and memory consistency models, this dissertation provides evidence showing that with efficient and scalable concurrency control mechanisms in runtime support, it is possible to get more reliable and scalable parallelism from shared-memory multicore systems and make the use of multicore for computations more accessible to the community.

8.2 Future Work

This section presents a brief overview of promising areas and ideas for the reader to consider working on.

Region serializability violation detection. Instead of detecting data races soundly during runtime, it is possible to detect data races that violate the underlying memory consistency model, e.g., region serializability, which are often most harmful. This insight can lead to data race detectors that add much lower overhead than traditional data race detectors. Researchers have investigated a plethora of techniques for detecting data races [57, 58, 61, 63, 26, 111]. However, existing data race detector often slows down programs by an order of magnitude. My coauthor and I have developed an approach called Valor to detect region conflicts using lazy read validation inspired by McRT's transactional conflict detection [17]. Valor significantly reduces the overhead to detect data races [17]. In particular, any data race can manifest as a region conflict [57]. Based on this, there are several interesting future directions. One direction is to investigate whether existing transactional memory techniques can be used for better design of region serializability violation detection and yield to lower overhead than Valor (which adds around 100% runtime execution overhead on average). Another direction is to design novel architectures that provide somewhat stronger consistency models. For example, instead of providing consistency at memory-access granularity [140], stronger consistency models could be achieved at the granularity of code regions (e.g., region serializability with fail-stop semantics). Furthermore, perhaps a slightly relaxed memory consistency model not as strong as region serializability would lead to better performance, scalability, and less hardware complexity.

Hardware/software hybrid TM with strong isolation and strong progress guarantees. Best-effort HTM needs to fall back to STM when hardware transactions fail. HTM naturally supports strong isolation, in which any conflicts caused by transactional or non-transactional accesses to shared memory locations will cause transactions to abort and to fallback to an STM solution. However, once HTM falls back to an STM, such strong isolation might no longer be guaranteed. Existing high-performance STMs mostly only support weak atomicity, so whatever transactional vs. transactional conflicts will be ignored by integrating a HTM and a weak STM system. Also, because of HTM limitations, software fallbacks are the only way to the TM system to guarantee progress. Our work on STM has demonstrated that it is possible to support strong isolation and strong progress guarantee with low-overhead, making it appealing for use in a hybrid TM that supports both strong isolation and strong progress guarantees, addressing one of the key limitations of HTM.

Providing SIx with multi-versioning concurrency control. Our Snappy approach ensures SIx with a variant of reader–writer lock that waits at a region end for a read–write conflict. Alternatively, one can support SIx with multi-versioning concurrency control (MVCC). Since under MVCC writers do not block readers, the availability of the SIx model could be further improved. MVCC ensures that each region operates against its own version (snapshot) which is created by finished regions. If a region R_i reads a variable `x`, the read operation is performed from R_i's snapshot, which should not be affected by concurrent regions. Writes are performed in-place on memory, and MVCC can use copy-on-write(CoW) to preserve the old version of memory that is being updated. On begin of a new region, MVCC assigns the region a unique timestamp (T.ts), which is from a

global timestamp that gets incremented each time a thread reaches a region boundary. Challenges include how to provide this global timestamp in an efficient and scalable way on a shared-memory system as well as how to modify MVCC to adapt to tolerating consistency exceptions.

Overall, there is still a lot of room for interesting research and solid contributions in the area of efficient runtime support for shared-memory multicore systems based on the work introduced in this dissertation. Our hope is that the direction and intuition presented here shed some light for future development in this area.

Bibliography

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *ACM Symposium on Principles of Programming Languages*, pages 63–74, 2008.

[2] Martín Abadi, Cormac Flanagan, and Stephen N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[3] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196, 2009.

[4] Sarita V. Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM*, 53:90–101, 2010.

[5] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1996.

[6] Sarita V. Adve and Mark D. Hill. Weak Ordering—A New Definition. In *ACM/IEEE International Symposium on Computer Architecture*, pages 2–14, 1990.

[7] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[8] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized Isolation Level Definitions. In *International Conference on Data Engineering*, pages 67–78, 2000.

[9] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and David Wong. BulkCompiler: High-performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *IEEE/ACM International Symposium on Microarchitecture*, pages 133–144, 2009.

[10] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[11] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *International Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.

[12] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *ACM/IEEE International Symposium on Computer Architecture*, pages 115–126, 2008.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995.

[14] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–64, 2010.

[15] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman, 1986.

[16] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *ACM Conference on Programming Language Design and Implementation*, pages 28–39, 2014.

[17] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 241–259, 2015.

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[19] Stephen M. Blackburn and Kathryn S. McKinley. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *ACM*

Conference on Programming Language Design and Implementation, pages 22–32, 2008.

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

[21] Hans-J. Boehm. How to miscompile programs with "benign" data races. In *USENIX Conference on Hot Topics in Parallelism*, 2011.

[22] Hans-J. Boehm. Position paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, pages 9–14, 2012.

[23] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *ACM Conference on Programming Language Design and Implementation*, pages 68–78, 2008.

[24] Hans-J. Boehm and Sarita V. Adve. You Don't Know Jack about Shared Variables or Memory Models. *Communications of the ACM*, 55(2):48–54, 2012.

[25] Hans-J. Boehm and Brian Demsky. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *ACM SIGPLAN Workshop on Memory System Performance and Correctness*, pages 7:1–7:6, 2014.

[26] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional Detection of Data Races. In *ACM Conference on Programming Language Design and Implementation*, pages 255–268, 2010.

[27] Michael D. Bond, Milind Kulkarni, Man Cao, Meisam Fathi Salmi, and Jipeng Huang. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *ACM International Conference on Principles and Practice of Programming in Java*, pages 90–101, 2015.

[28] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 693–712, 2013.

[29] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, 2002.

[30] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *ACM Symposium on Operating Systems Principles*, pages 1–11, 1995.

[31] Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Feedback-Directed Barrier Optimization in a Strongly Isolated STM. In *ACM Symposium on Principles of Programming Languages*, pages 213–225, 2009.

[32] Mike Burrows. How to Implement Unnecessary Mutexes. In *Computer Systems Theory, Technology, and Applications*, pages 51–57. Springer–Verlag, 2004.

[33] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 187–200, 2014.

[34] Man Cao, Minjia Zhang, and Michael D. Bond. Drinking from Both Glasses: Adaptively Combining Pessimistic and Optimistic Synchronization for Efficient Parallel Runtime Support. In *Workshop on Determinism and Correctness in Parallel Programming*, 2014.

[35] Man Cao, Minjia Zhang, Aritra Sengupta, and Michael D. Bond. Drinking from Both Glasses: Combining Pessimistic and Optimistic Tracking of Cross-Thread Dependences. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 20:1–20:13, 2016.

[36] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IEEE International Symposium on Workload Characterization*, 2008.

[37] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Communications of the ACM*, 51(11):40–46, 2008.

[38] Luis Ceze, Joseph Devietti, Brandon Lucia, and Shaz Qadeer. A Case for System Support for Concurrency Exceptions. In *USENIX Conference on Hot Topics in Parallelism*, 2009.

[39] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

[40] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[41] Mark Christiaens and Koenraad De Bosschere. Accordion Clocks: Logical Clocks for Data Race Detection. In *International European Conference on Parallel Processing*, pages 494–503, 2001.

[42] Mark Christiaens and Koen De Bosschere. TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs. In *Symposium on Java Virtual Machine Research and Technology Symposium*, pages 15–15, 2001.

[43] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *European Conference on Computer Systems*, pages 27–40, 2010.

[44] Cliff Click. Azul's Experiences with Hardware Transactional Memory. In HP Labs – Bay Area Workshop on Transactional Memory, January 2009.

[45] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 11. The MIT Press, McGraw-Hill Book Company, 2nd edition, 2001.

[46] Luke Dalessandro and Michael L. Scott. Strong Isolation is a Weak Idea. In *ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[47] Luke Dalessandro and Michael L. Scott. Sandboxing Transactional Memory. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 171–180, 2012.

[48] Luke Dalessandro, Michael L. Scott, and Michael F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *International Symposium on Distributed Computing*, pages 20–34, 2010.

[49] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 67–78, 2010.

[50] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent Code Generation: Implementation, Analysis, and Evaluation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 1–12, 2013.

[51] Brian Demsky and Alokika Dash. Evaluating Contention Management Using Discrete Event Simulation. In *ACM SIGPLAN Workshop on Transactional Computing*, 2010.

[52] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2009.

[53] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.

[54] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *International Symposium on Distributed Computing*, pages 194–208, 2006.

[55] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM Can Be More than a Research Toy. *Communications of the ACM*, 54:70–77, 2011.

[56] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching Transactional Memory. In *ACM Conference on Programming Language Design and Implementation*, pages 155–165, 2009.

[57] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 467–484, 2012.

[58] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *ACM Conference on Programming Language Design and Implementation*, pages 245–255, 2007.

[59] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database Replication Using Generalized Snapshot Isolation. pages 73–84, 2005.

[60] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.

[61] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, 2010.

[62] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005.

[63] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *ACM Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[64] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 293–303, 2008.

[65] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: A Portable Record/Replay Environment for Multi-threaded Java Applications. *Software Practice & Experience*, 34(6):523–547, 2004.

[66] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 57–76, 2007.

[67] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared-memory Multiprocessors. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, 1991.

[68] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An Exploratory Survey. In *Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[69] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 101–110, 2010.

[70] Dan Grossman. The Transactional Memory / Garbage Collection Analogy. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 695–706, 2007.

[71] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a Theory of Transactional Contention Managers. In *ACM Symposium on Principles of Distributed Computing*, pages 258–264, 2005.

[72] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *ACM/IEEE International Symposium on Computer Architecture*, pages 102–113, 2004.

[73] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[74] Tim Harris and Keir Fraser. Revocable Locks for Non-Blocking Programming. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 72–82, 2005.

[75] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[76] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing Memory Transactions. In *ACM Conference on Programming Language Design and Implementation*, pages 14–25, 2006.

[77] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Remote Invalidation: Optimizing the Critical Path of Memory Transactions. In *IEEE International Parallel & Distributed Processing Symposium*, pages 187–197, 2014.

[78] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. In *ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[79] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ACM/IEEE International Symposium on Computer Architecture*, pages 289–300, 1993.

[80] Benjamin Hindman and Dan Grossman. Atomicity via Source-to-Source Translation. In *ACM SIGPLAN Workshop on Memory System Performance and Correctness*, pages 82–91, 2006.

[81] Baris Kasikci, Cristian Zamfir, and George Candea. RaceMob: Crowdsourced Data Race Detection. In *ACM Symposium on Operating Systems Principles*, pages 406–422, 2013.

[82] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–141, 2002.

[83] Edgar Knapp. Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[84] Guy Korland, Nir Shavit, and Pascal Felber. Deuce: Noninvasive Software Transactional Memory in Java. *Transactions on HiPEAC*, 5(2), 2010.

[85] Ismail Kuru, Burcu Kulahcioglu Ozkan, Suha Orhun Mutluergil, Serdar Tasiran, Tayfun Elmas, and Ernie Cohen. Verifying Programs under Snapshot Isolation and Similar Relaxed Consistency Models. In *ACM SIGPLAN Workshop on Transactional Computing*, 2014.

[86] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Computer*, 28:690–691, 1979.

[87] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[88] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36:471–482, 1987.

[89] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *ACM Conference on Programming Language Design and Implementation*, pages 463–474, 2012.

[90] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–90, 2010.

[91] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[92] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient Sequential Consistency Using Conditional Fences. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 295–306, 2010.

[93] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 273–286, 2012.

[94] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. SI-TM: Reducing Transactional Memory Abort Rates Through Snapshot Isolation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 383–398, 2014.

[95] Heiner Litz, Ricardo J. Dias, and David R. Cheriton. Efficient Correction of Anomalies in Snapshot Isolation Transactions. 11(4):65:1–65:24, 2015.

[96] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient Deterministic Multithreading. In *ACM Symposium on Operating Systems Principles*, pages 327–336, 2011.

[97] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[98] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ACM/IEEE International Symposium on Computer Architecture*, pages 210–221, 2010.

[99] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ACM/IEEE International Symposium on Computer Architecture*, pages 277–288, 2008.

[100] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *ACM Symposium on Principles of Programming Languages*, pages 378–391, 2005.

[101] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *ACM Conference on Programming Language Design and Implementation*, pages 351–362, 2010.

[102] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A Case for an SC-Preserving Compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 199–210, 2011.

[103] Hassan Salehe Matar, Ismail Kuru, Serdar Tasiran, and Roman Dementiev. Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support. In *Workshop on Determinism and Correctness in Parallel Programming*, 2014.

[104] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical Weak-Atomicity Semantics for Java STM. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 314–325, 2008.

[105] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single Global Lock Semantics in a Weakly Atomic STM. In *ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[106] Katherine F. Moore and Dan Grossman. High-Level Small-Step Operational Semantics for Transactions. In *ACM Symposium on Principles of Programming Languages*, pages 51–62, 2008.

[107] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006.

[108] Mayur Naik and Alex Aiken. Conditional Must Not Aliasing for Static Race Detection. In *ACM Symposium on Principles of Programming Languages*, pages 327–338, 2007.

[109] Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[110] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware Atomicity for Reliable Software Speculation. In *ACM/IEEE International Symposium on Computer Architecture*, pages 174–185, 2007.

[111] Robert O'Callahan and Jong-Deok Choi. Hybrid Dynamic Data Race Detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[112] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–108, 2009.

[113] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 365–375, 2007.

[114] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. ...and region serializability for all. In *USENIX Conference on Hot Topics in Parallelism*, 2013.

[115] Victor Pankratius and Ali-Reza Adl-Tabatabai. A Study of Transactional Memory vs. Locks in Practice. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 43–52, 2011.

[116] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.

[117] PCWorld. Nasdaq's facebook glitch came from race conditions, 2012. `http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[118] Dan R. K. Ports and Kevin Grittner. Serializable Snapshot Isolation in PostgreSQL. 5(12):1850–1861, 2012.

[119] Eli Pozniansky and Assaf Schuster. MultiRace: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *Concurrency and Computation: Practice & Experience*, 19(3):327–340, 2007.

[120] Parthasarathy Ranganathan, Vijay Pai, and Sarita Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *ACM Symposium on Parallelism in Algorithms and Architectures*, page pages, 1997.

[121] James Reinders. Transactional Synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`.

[122] Carl G. Ritson and Frederick R.M. Barnes. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *Communicating Process Architectures*, pages 271–292, 2013.

[123] Kenneth Russell and David Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 263–272, 2006.

[124] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, 2006.

[125] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.

[126] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, 1996.

[127] William N. Scherer III and Michael L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[128] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Dynamic Optimization for Efficient Strong Atomicity. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 181–194, 2008.

[129] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, 1994.

[130] Cedomir Segulja and Tarek S. Abdelrahman. Clean: A Race Detector with Cleaner Semantics. In *ACM/IEEE International Symposium on Computer Architecture*, pages 401–413, 2015.

[131] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 561–575, 2015.

[132] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer – data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009.

[133] Jaroslav Ševčík and David Aspinall. On Validity of Program Transformations in the Java Memory Model. In *European Conference on Object-Oriented Programming*, pages 27–51, 2008.

[134] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, 1988.

[135] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *ACM Conference on Programming Language Design and Implementation*, pages 78–88, 2007.

[136] João M Silva, José Simão, and Luís Veiga. Ditto – Deterministic Execution Replayability-as-a-Service for Java VM on Multiprocessors. In *ACM/IFIP/USENIX International Middleware Conference*, pages 405–424, 2013.

[137] Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–66, 2011.

[138] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-End Sequential Consistency. In *ACM/IEEE International Symposium on Computer Architecture*, pages 524–535, 2012.

[139] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *ACM/IEEE Supercomputing Conference*, pages 8–8, 2001.

[140] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. 1992.

[141] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 141–150, 2009.

[142] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization Techniques for Software Transactional Memory. In *ACM Symposium on Principles of Distributed Computing*, 2007.

[143] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–284, 2008.

[144] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 2–13, 2005.

[145] Herb Sutter. The Free Lunch Is Over , 2005.

[146] U.S.–Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada. Technical report, Department of Energy, 2004.

[147] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2009.

[148] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26, 2011.

[149] Christoph von Praun and Thomas R. Gross. Object Race Detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

[150] Christoph von Praun and Thomas R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 115–128, 2003.

[151] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–214, 2007.

[152] Jons-Tobias Wamhoff, Christof Fetzer, Pascal Felber, Etienne Rivière, and Gilles Muller. FastLane: Improving Performance of Software Transactional Memory for Low Thread Counts. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 113–122, 2013.

[153] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136, 2012.

[154] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 34–48, 2007.

[155] Liqiang Wang and Scott D. Stoller. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 137–146, 2006.

[156] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Low-Level Detection of Language-Level Data Races with LARD. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 671–686, 2014.

[157] Peng Wu, Maged M. Michael, Christoph von Praun, Takuya Nakaike, Rajesh Bordawekar, Harold W. Cain, Calin Cascaval, Siddhartha Chatterjee, Stefanie Chiras, Rui Hou, Mark Mergen, Xiaowei Shen, Michael F. Spear, Hua Yong Wang, and Kun Wang. Compiler and Runtime Techniques for Software Transactional Memory Optimization. *Concurrency and Computation: Practice and Experience*, 21(1):7–23, 2009.

[158] Zhemin Yang, Min Yang, Lvcai Xu, Haibo Chen, and Binyu Zang. ORDER: Object Centric Deterministic Replay for Java. In *USENIX Annual Technical Conference*, pages 30–30, 2011.

[159] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *ACM/IEEE Supercomputing Conference*, pages 19:1–19:11, 2013.

[160] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, 2008.

[161] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *ACM Symposium on Operating Systems Principles*, pages 221–234, 2005.

[162] Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. All That Glitters is Not Gold: Improving Availability and Practicality of Exception-Based Memory Models. Technical Report OSU-CISRC-4/16-TR01, Computer Science & Engineering, Ohio State University, 2016.

[163] Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. Relaxed Dependence Tracking for Parallel Runtime Support. In *International Conference on Compiler Construction*, CC 2016, pages 45–55, 2016.

[164] Minjia Zhang, Jipeng Huang, Man Cao, and Michael D. Bond. Low-Overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 97–108, 2015.

[165] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *International Symposium on High-Performance Computer Architecture*, pages 121–132, 2007.

[166] Craig Zilles and Ravi Rajwar. Transactional Memory and the Birthday Paradox. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 303–304, 2007.

[167] Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. Discovering and Understanding Performance Bottlenecks in Transactional Applications. In *IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 285–294, 2010.