Practical Support for Strong, Serializability-Based Memory Consistency

Dissertation

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

By

Swarnendu Biswas, B.E., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Dissertation Committee:

Michael D. Bond, Advisor

Atanas Rountev

Feng Qin

Abstract

Data races are a fundamental barrier to writing correct shared-memory, multithreaded programs, and often lead to concurrency bugs. Furthermore, data races complicate programming language semantics. Current programming language and hardware memory models provide *weak end-to-end consistency guarantees for executions with data races*—leading to unexpected, erroneous behaviors.

Researchers have argued for a stronger guarantee in the presence of concurrency errors arising from data races: *programs should have fail-stop behavior and precise semantics*. However, building a system that efficiently provides strong semantic guarantees for data races is complex and challenging—and has remained elusive.

The complexity and risk associated with data races motivate this work. The thesis of our work is that systems should furnish programming language implementations with a mechanism that gives all executions clear, simple semantics. In this dissertation, we advocate for providing *strong memory consistency based on serializability of regions* as the default memory model on shared-memory systems. Prior approaches that have attempted to provide region serializability were afflicted by the high costs of either precisely tracking "last reader(s)" information at read operations or eagerly tracking all conflicts. To realize this goal, we explore efficient techniques to provide strong memory consistency based on region serializability—where regions are demarcated by synchronization operations, or they can be programmer-defined. We begin by developing a software-only technique that provides

serializability of *synchronization-free regions* (i.e., regions demarcated by synchronization operations) by efficiently detecting region conflicts. For a given execution, our proposed technique ensures that the execution either completes successfully and the output is region serializable, or the execution terminates with an exception. The key insight in this work is that detecting read–write conflicts lazily retains necessary semantic guarantees and has better performance than eager conflict detection. Our proposed software-only technique has overheads competitive enough to provide practical semantic guarantees to a language specification. Since hardware support can speed up conflict detection, we then explore the possibility of an efficient architectural solution to provide region serializability end-to-end. We propose a novel architecture design that provides the same strong consistency guarantee for every execution by either providing region serializability or by generating a consistency exception indicating a data race that may jeopardize consistency. The key insight in our hardware design is that each core can execute largely independently from other cores, deferring actions that ensure consistency until synchronization operations and private cache evictions. The key contributions lie in its novel mechanism to detect region conflicts. Furthermore, as a result of ensuring consistency, the design can defer cache coherence until synchronization operations and evictions—unlike existing coherence protocols that ensure coherence at every instruction. Lastly, we extend the scope of providing consistency to the language-level by providing serializability of programmer-defined regions instead of synchronization-free regions. Serializability of programmer-defined regions or atomicity is a key correctness property of concurrent programs that allows programmers to reason about code regions in isolation. However, programs often fail to enforce atomicity correctly, leading to serializability violations that are difficult to detect. We present a novel sound and precise dynamic atomicity checker that *checks* for violations of conflict serializability

efficiently. The key insight of this work lies in soundly and efficiently overapproximating cross-thread dependences, and then recovering precision with more expensive analysis only when required.

Even after years of research, providing strong memory consistency based on serializability of regions has so far remained elusive. As a case in point: current high-level programming languages such as C/C++ and Java memory models provide essentially no useful guarantees about the semantics of data races. This dissertation aims to fill this critical gap in today's languages and systems by demonstrating the possibility of providing a strong memory model based on region serializability efficiently. By precisely identifying potential serializability violations, the approaches proposed in this dissertation help extend strong semantic guarantees to *all* program executions, including executions with data races. By precisely checking for atomicity violations, our technique presents a promising direction for providing serializability of programmer-defined regions. The software-only performance-efficient techniques presented in this work can provide the foundation to guarantee stronger semantics to all program executions in the near-future, thereby improving concurrent software development and reliability in the long-term. Our evaluation shows that the hardware-based solution can be the foundation upon which future scalable shared-memory systems are built. Overall, this dissertation significantly advances the state of the art in parallel architecture consistency and coherence.

Dedicated to my parents and my wife

Acknowledgments

There are many people who have directly or indirectly contributed to this dissertation. I am grateful to all of them.

It has been a wonderful and enriching experience to work with my advisor Dr. Michael D. Bond at the Ohio State University. I guess it was just meant to be, since I did not know him when I applied to the graduate school. Everything fell into place once Mike contacted me to see if I was interested in working with him. During all these years, he has been very supportive and encouraging in every aspect of life, both professional and personal. I am grateful to him for all his contributions.

I have had the good fortune to interact with several learned people both in and out of the university. Dr. Brandon Lucia from Carnegie Mellon has been a collaborator on some of my projects, and his expertise in architecture and simulation has been of great help. He is fun to work with. Prof. Atanas Rountev and Feng Qin have been on my reading committee and I thank them for their advice and feedback. I would like to acknowledge the support of Prof. P. Sadayappan and other professors in our department.

I thank the staff members of the department for their continued help, and am also grateful for the wonderful facilities provided by the department.

Successful projects generally result from contributions from several collaborators—they are seldom the work of one person. I would like to acknowledge all the help from the

PLaSS group members: Man Cao, Meisam Fathi-Salmi, Jipeng Huang, Jake Roemer, Aritra Sengupta, Minjia Zhang, and Rui Zhang. They are an awesome bunch of people.

Columbus is a wonderful city to live in, and I thoroughly enjoyed my stay here. Nonetheless, my experience at Columbus was all the more fun-filled and memorable because of the quality time spent with friends. I would like to acknowledge the companionship of Bortik Bandyopadhyay, Arka Bhattacharya, Samik Bhattacharya, Aniket Chakrabarti, Ashesh Choudhury, Soumya Dutta, Rajaditya Mukherjee, Dhrubojyoti Roy, and Aritra Sengputa.

It would have been impossible to complete my PhD journey and achieve whatever little I have today without the unwavering support and sacrifice of my parents. They have encouraged me, guided me and believed in me every step of my life. I am grateful to them for ever.

I met my lovely wife during my PhD program. She is a wonderful companion, I am grateful to her for her understanding and constant support.

The material presented in this dissertation is based upon work supported by the National Science Foundation under generous grants CSR-1218695, CAREER-1253703, and CCF-1421612. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Vita

August 2016	PhD, Computer Science and Engineering, The Ohio State University, USA.
May 2015	M.S., Computer Science and Engineering, The Ohio State University, USA.
August 2011	M.S., Computer Science and Engineering, IIT Kharagpur, India.
May 2005	B.E., Computer Science and Engineering, NIT Durgapur, India.

Publications

Research Publications

Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. All That Glitters Is Not Gold: Improving Availability and Practicality of Exception-Based Memory Models. In *Ohio State CSE Technical Report* #OSU-CISRC-4/16-TR01, April 2016.

Minjia Zhang, Swarnendu Biswas, and Michael D. Bond. Relaxed Dependence Tracking for Parallel Runtime Support. In *International Conference on Compiler Construction*, pages 45–55, March 2016.

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

Swarnendu Biswas. Viser: Providing Serializability in Hardware with Simplified Cache Coherence. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 75–76, October 2015.

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

Swarnendu Biswas, Minjia Zhang, and Michael D. Bond. Lightweight Data Race Detection for Production Runs. In *Ohio State CSE Technical Report* #OSU-CISRC-1/15-TR01, January 2015.

Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 28–39, June 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 *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 693-712, March 2013.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Programming Languages and Software Systems	Prof. Michael D. Bond
High Performance Computing	Prof. P. Sadayappan
Theory and Algorithms	Prof. Anish Arora

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Current multicore hardware trends of providing more—instead of faster—cores have made parallelism necessary for performance. As multicore systems become widespread, software and hardware face a growing challenge in efficiently implementing and exploiting parallelism. For example, programmers are now hard-pressed to develop correct and high-performance concurrent programs to exploit every iota of performance available from multicore systems.

The shared memory paradigm is the most widely-used programming model and the *de facto* standard for programming parallel applications. Shared-memory multiprocessors enable an execution model that is simple and efficient: multiple threads execute on processor cores that share a global memory. In other words, the shared-memory programming model simplifies parallel programming by providing an address space shared by different threads of execution potentially running on different processor cores.

1.1 Motivation and Problem Statement

Recent studies and experiences have shown that developing shared-memory parallel programs that are correct and scalable is notoriously challenging [80, 112]. David Patterson explained the seriousness of this issue in a 2006 interview [137]:

> *From my perspective, parallelism is the biggest challenge since high-level programming languages. It's the biggest thing in 50 years because industry is betting its future that parallel programming will be useful.*
>
> *Industry is building parallel hardware, assuming people can use it. And I think there's a chance they'll fail since the software is not necessarily in place. So this is a gigantic challenge facing the computer science community. If we miss this opportunity, it's going to be bad for the industry.*

A key challenge in writing and debugging shared-memory parallel programs is that programmers must reason about many possible behaviors because of the many ways that threads' memory accesses can interleave. The *bête noire* of shared memory concurrent programming is a *data race* which occurs when multiple threads access the same memory location without synchronization (Section 2.1). A fundamental barrier to achieving correctness with shared-memory systems is that shared-memory programming languages and architectures provide weak or undefined semantics for executions with data races [2, 8, 31, 118, 157, 164, 174]—leading to unexpected, erroneous behaviors [2, 29, 30, 32, 33, 112].

For performance reasons, shared-memory programming language compilers and runtimes employ several optimizations that were developed and are provably correct in the single-threaded context. Building a shared-memory system that *efficiently* provides strong semantic guarantees for data races is complex and challenging—and has remained elusive even after years of research. Instead, limiting clear, intuitive semantics to *programs that are free of data races* allow these language compilers and runtimes to continue to use existing optimizations. As a case in point: languages such as C/C++ and Java provide essentially no useful guarantees about the semantics of data races [2, 5, 29, 30, 115, 163] (Section 2.2.1).

According to Adve and Boehm [2], "The memory model defines an interface between a program and any hardware or software that may transform that program (for example, the compiler, the virtual machine, or any dynamic optimizer)." While language memory models provide virtually no guarantees for executions with data races [2, 31, 118], nearly

all architectures enforce *comparatively strong* guarantees for program executions with data races. Still, pervasive hardware memory models used by x86 [164], SPARC [174], Power [157], and ARM [8] are weak, permitting visible reordering of memory accesses [8, 157, 164, 174]. Furthermore, hardware memory models are thus not only weak, but their guarantees are *not end-to-end*, i.e., they are with respect to the compiled program only.

Problem statement. *The complexity and risk associated with data races motivate this work.* The thesis of our work is that systems should provide strong memory models end-to-end. That is, programming language implementations or systems should provide data races with clear, simple semantics. *Our goal in this work is to develop mechanisms that equip present and future shared-memory languages and systems with clear, intuitive semantics even for programs that permit data races.*

1.2 Proposed Solutions for Practical Support for Strong, Serializability-Based Memory Consistency

Current shared-memory languages and systems provide a memory model, called *SFR serializability*, for data-race-free (DRF) programs [5, 31, 115, 118, 139]. In the absence of data races, all program operations are correctly synchronized, and hence synchronization-free regions (SFRs) of code—which are dynamic regions of code between synchronization operations—appear to execute atomically (i.e., without region conflicts). SFR serializability is a strong memory model as it allows few interleavings. Another advantage of SFR serializability is that compiler and hardware optimizations are free to reorder memory accesses within SFRs. Furthermore, architecture support alone can ensure end-to-end SFR serializability because compilers already respect synchronization boundaries. *These advantages make SFR serializability an appealing memory model, and it extends the same*

guarantees to executions with races that current languages and systems give to only race-free executions. We discuss SFR serializability in detail in Section 2.2.2.

In the following, we briefly describe our contributions to efficiently provide a strong memory model called *SFRSx*, which is based on serializability of regions. Even in an execution with data races, SFRSx guarantees the serializability of SFRs of code, or halts with a consistency exception that indicates a conflict. We first describe a software-only technique that provides SFR serializability as a strong memory consistency model. Since conflict detection in hardware can be more efficient, we provide end-to-end region serializability with architectural modifications. Lastly, we extend the scope of providing consistency to the language-level by providing serializability of programmer-defined regions.

1.2.1 Providing SFR Serializability in Software

We present two new software-based region conflict detectors, called *FastRCD* and *Valor*, that provide SFR serializability [20]. FastRCD is a new dynamic analysis for detecting region conflicts that leverages the epoch optimization strategy of the current state-of-the-art data race detector [68]. We have developed FastRCD more for the purposes of comparison, and show that it continues to suffer from performance bottlenecks similar to prior work. The primary contribution of the work is Valor, which is a sound and precise region conflict detection analysis that achieves high performance by eliminating the costly analysis on each read operation that prior approaches (and FastRCD) require. Valor instead logs a region's reads and lazily detects conflicts for logged reads when the region ends. A successful program execution with Valor indicates that the execution was region (e.g., SFR) serializable, i.e., all regions executed atomically, otherwise Valor raises an exception. Valor

is the first region conflict detector to provide strong semantic guarantees for racy program executions with under 2X slowdown.

1.2.2 Providing SFR Serializability in Hardware

To provide strong memory models, one must consider implications for the whole system [2]. For example, "a processor vendor cannot guarantee a strong hardware model if the memory system designer provides a weaker model; a strong hardware model is not very useful to programmers using languages and compilers that provide only a weak guarantee" [2]. Furthermore, since conflict detection can be cheaper with hardware support, in this work, we explore a hardware solution to provide strong memory consistency, i.e., SFRSx, end-to-end. We propose a novel architecture design called *Region Consistency and Coherence* that provides a strong consistency guarantee: for every execution, RCC either provides serializability of synchronization-free regions (SFRs), or it generates a consistency exception indicating a data race that may jeopardize consistency. RCC's insight is that each core can execute largely independently from other cores, deferring actions that ensure consistency until synchronization operations and private cache evictions. Furthermore, as a result of ensuring consistency, RCC can defer cache coherence until synchronization operations and evictions—unlike existing coherence protocols that ensure coherence at every instruction. RCC's contribution is a novel mechanism for detecting conflicts by ensuring write atomicity and checking read consistency. RCC eliminates the need for legacy coherence protocol support, making better use of hardware resources, yielding a design that is both performant and complexity-effective. Our evaluation shows that RCC compares favorably with current shared-memory implementations. Furthermore, RCC has clear advantages in terms of implementability over prior work called Conflict Exceptions (CE) and TCC, in terms of run-time

performance, on-chip and off-chip traffic. While an unoptimized design of RCC incurs significant costs to provide coherence conservatively, we show how optimizations reduce these costs substantially. Compared to Valor, RCC provides end-to-end SFR serializability *more efficiently but at the cost of hardware modifications*.

1.2.3 Checking Serializability of Programmer-Defined Regions

Instead of limiting the dissertation's scope to detecting serializability violations of regions demarcated by synchronization operations, this work checks for serializability violations of programmer-defined regions. We propose a novel dynamic analysis to check for violations of serializability, which is also known as the atomicity in programming languages. Dynamic program analysis can detect atomicity violations in concurrent programs based on an atomicity specification, but existing approaches incur large overheads [67, 73, 113, 185]. Our technique, called *DoubleChecker*, is an efficient, sound and precise dynamic atomicity violation detector [18] that substantially outperforms current state-of-art [73]. The key insight of DoubleChecker lies in its use of two new staged and cooperating dynamic analyses—an *imprecise* and a *precise* analysis. Its imprecise analysis tracks cross-thread dependences soundly but imprecisely with significantly better performance than a fully precise analysis. Its precise analysis is more expensive but only needs to process a subset of the execution identified as potentially involved in atomicity violations by the imprecise analysis. DoubleChecker can operate in two modes. In *single-run* mode, the two analyses execute in the same program run, which guarantees soundness and precision but requires logging program accesses to pass from the imprecise to the precise analysis. In *multi-run* mode, the first program run executes only the imprecise analysis, and a second run executes both analyses. Multi-run mode trades accuracy for performance; each run of multi-run

mode outperforms single-run mode, but can potentially miss violations. DoubleChecker is a promising direction for improving the performance of dynamic atomicity checking over prior work.

1.3 Contributions and Impact

The memory consistency model is considered to be fundamental to the concurrency semantics of a shared-memory program or system [2]. Current shared-memory language and hardware memory models are complicated by the presence of data races, and do not provide strong semantics for racy program executions which can and *has* led to erroneous behaviors [107, 145, 179]. Furthermore, complex memory models make it difficult to learn and write correct parallel programs. Alternate memory models are either too strict (i.e., allows no or few reorderings limiting hardware and compiler optimizations) severely reducing performance, or are too relaxed and places the burden of correctness on the programmer (Section 2.2).

To deal with data races, current languages and systems need to adopt stronger memory models. One such desired strong memory model is SFR serializability, which extends to executions with data races the same end-to-end guarantees that today's language and hardware memory models provide but only for DRF executions [2]. So far, efficient techniques for providing SFR serializability have remained elusive. In this dissertation, we show that achieving strong memory consistency with serializability of SFRs is practical. We propose efficient and realizable techniques that advance the state of the art in *providing* strong memory consistency based on region serializability. For example, Valor advances the state of the art in always-on software support for strong behavioral guarantees for data races. DoubleChecker presents a new direction in dynamic sound and precise atomicity checking

and dynamic analyses (e.g., other types dynamic concurrency bug detection such as data race detection [19]) in general: by dividing the analysis into two staged and cooperating analyses, our work shows that it is often beneficial to over-approximate dependences and recover precision on demand. The improved performance overhead of atomicity checking presented in this proposal, coupled with the fact that our proposed solution is software-only, suggests that the technique proposed in this work represent practical, promising direction for improving the performance of concurrency bug detectors and making it more widely available. The software-only techniques presented in this dissertation, *Valor* and *DoubleChecker*, have the potential to be integrated into future language runtimes to provide better software reliability. For example, a language runtime can integrate Valor to provide *always-on* support for strong behavioral guarantees for data races, and DoubleChecker to warn developers about potential atomicity violations. Such support provided by runtime systems will help developers identify and eliminate most or all data races that lead to serializability violations and atomicity violations during testing, and will terminate program executions on production systems before allowing bad behavior to actually happen. RCC does require extensions to the current architecture, but our evaluation shows that it can be the foundation upon which future scalable shared-memory systems are built. Since RCC provides significantly stronger guarantees than current shared-memory systems at a comparable cost, it is a compelling design for providing consistency and coherence in future systems.

As acknowledged by Adve and Boehm [2], strengthening the memory model decision can have a long-lasting impact on the systems community, affecting portability and maintainability of programs. This dissertation contributes to the cause.

1.4 Outline

In the following, we outline the work presented in this dissertation.

- We discuss background on data race detection, memory models, and atomicity checking along with closely related work in **Chapter 2**.
- We present a software-only solution called *Valor* [20] to provide SFR serializability in **Chapter 3**.
- We present an architectural solution called *RCC* to provide SFR serializability in **Chapter 4**.
- We present our work on efficient, sound and precise dynamic atomicity violation detection technique called *DoubleChecker* [18] in **Chapter 5**.
- We discuss other related work in **Chapter 6**.
- We discuss possible improvements to our proposed techniques and potential future work in **Chapter 7**.
- We conclude our dissertation and discuss the impact of our work in **Chapter 8**.

Chapter 2: Background and Related Work

Today, software is omnipresent. Our society is increasingly reliant on software to solve tasks that range from simple household applications to more critical tasks in areas such as medicine, engineering, transportation, and energy. With the advent of the multicore era, computing systems must become more parallel and at the same time more reliable in order to address future unsolved problems.

Software bugs are the primary cause of failures in production systems compared to hardware failures [11]. It is estimated that software companies spend over 30% of their development resources on software testing, but even then it is difficult to guarantee bug-free software due to several challenges. As a result, software bugs—both sequential and concurrent—often escape into production systems. Concurrency bugs constitute a major chunk of these software errors [155]. The Therac-25 medical accident [107], the Northeastern electricity blackout [179], and the Nasdaq Facebook glitch [145] are a few examples of several high-profile failures, and bear testament to the fact that software errors (including concurrency bugs) are present even in well-tested code (e.g., [74, 112]). According to the report of National Institute of Standards and Technology (NIST) in 2002, software errors caused the US economy an estimated 59.5 billion annually, or about 0.6 percent of the gross domestic product (GDP). In short, the impact of software and concurrency bugs is *enormous*.

Testing of concurrent programs is getting even more challenging in the current multicore era. The increased state space of programs involving multiple threads and shared memory makes it extremely difficult to detect all possible bugs during testing. Moreover, diagnosing concurrency errors is in itself challenging, given the nondeterministic nature of concurrent programs. Studies suggest that it often takes developers weeks or months to diagnose and fix concurrency issues [80, 112], and even then risk introducing additional bugs as part of the fix itself [68, 96]. Concurrency bugs will only become more problematic as software systems become increasingly parallel to scale with parallel hardware.

Concurrency bugs can be of different types such as atomicity violations, order violations, data races, and deadlocks [112]. The more common sources of non-deadlock concurrency bugs among others are due to *data races* and *atomicity violations*. There has been lot of research on trying to automatically detect concurrency bugs with program analyses to help improve programmer productivity and software reliability. A plethora of static and dynamic analyses have been proposed to detect different types of concurrency bugs such as atomicity violations [63,67,72–74,113,185], data races [34,56,60,68,94,99,120,131,132,136,146,170], and deadlocks [97, 186]. However, existing concurrency bug detection techniques tend to suffer from a variety of drawbacks, ranging from performance overheads, correctness and precision concerns, scalability, and/or custom hardware support, which have affected the practical adoption of such techniques. For example, static analysis techniques have the potential to be sound and complete (i.e., generate no false negatives), but these techniques usually suffer from poor precision (i.e., report false positives) (e.g., [132]) and do not scale well to large real-world programs (e.g., [131]). Existing dynamic analyses usually incur high performance overheads to guarantee both soundness and precision [68, 73], and often resort to making a tradeoff to balance all desired features [56, 158]. These performance

concerns have affected the usability of existing dynamic analysis techniques. Developers desist using tools that report false positives due to the possibility of wasted work [34, 120], whereas programmers and testers shy away from using intrusive tools that do not allow them to test realistic program executions [120]. No wonder the practical adoption of these existing techniques is still muted! *Low-overhead* checking is needed in order to use it liberally to find bugs during in-house, alpha, and beta testing, and perhaps even some production settings. Greathouse et al. [81] note that high dynamic analysis overheads "reduce the degree to which programs can be tested within a reasonable amount of time. Beyond that, high overheads slow debugging efforts, as repeated runs of the program to hunt for root causes and verify fixes also suffer these slowdowns."

In this chapter, we first present some background on data races and their impact on programming language memory consistency models. We then conclude this chapter by discussing about atomicity checking. Atomicity, which is synonymous to serializability in programming languages, is a fundamental correctness property in concurrent programs. Unfortunately, programs often fail to correctly enforce atomicity, and atomicity violations are the primary source of concurrency bugs.

2.1 Data Races

Data races often indicate presence of other sources of non-deadlock concurrency bugs such as order violations.

2.1.1 Definitions

Definition 2.1. The *happens-before* relation ($\prec_{HB}$) is a transitively-closed, irreflexive, partial order relation on the operations in an execution [34, 68, 104]. A dynamic statement A happens-before B (i.e., $A \prec_{HB} B$) if any of the following conditions hold:

```
MyObject x = null;
boolean done = false;

Thread 1                        Thread 2

x = new MyObject();             while (!done);
done = true;                    x.compute();
```

Figure 2.1: A Java code snippet containing data races on variables x and done.

- PROGRAM ORDER: Statement A executes before B in the same thread.
- SYNCHRONIZATION ORDER: Statement A and B are operations on the same synchronization variable such that the semantics imply a happens-before edge (e.g., A releases a lock, and B subsequently acquires the same lock).
- TRANSITIVE RELATION: $A \prec_{HB} C$ and $C \prec_{HB} B$.

If A happens before B, then it is also the case that B *happens after* A.

Definition 2.2. Two operations in an execution are *concurrent* if they are not related by the happens-before relation.

Definition 2.3. Two memory accesses from different threads are said to be *conflicting* if they both access (read or write) the same variable, and at least one of the operations is a write.

Definition 2.4. In a concurrent program execution, a *data race* occurs when two accesses are conflicting—executed by different threads and at least one is a write—and concurrent—not ordered by synchronization operations [6].

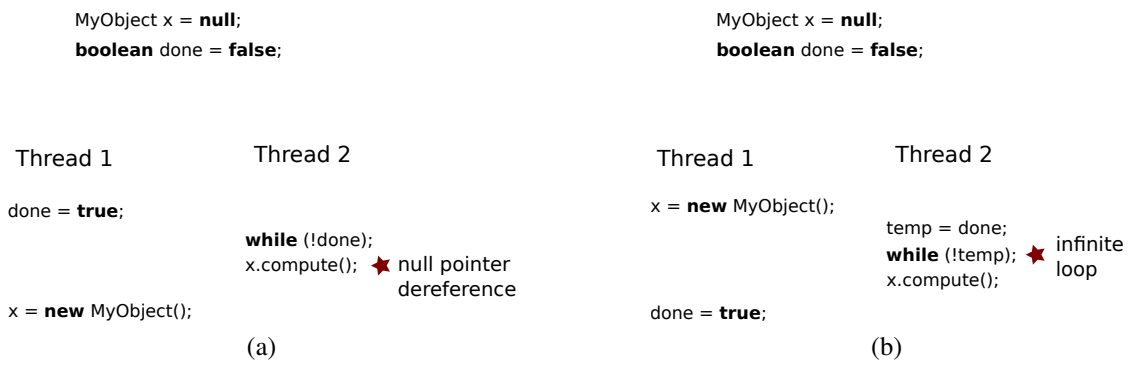


Figure 2.2: Transformations on a racy program can lead to surprising behaviors.

Figure 2.1 shows a simple but often used Java code where two threads try to incorrectly coordinate their execution via the shared variables x and done. The accesses to x and done are racy since no happens-before relation exists between the dynamic program statements accessing the variables (i.e., none of the three conditions listed in Definition 2.1 hold).

Data races can cause programs to exhibit confusing and incorrect behavior. In an attempt to improve performance, the compiler and hardware can reorder instructions in each thread independently of other concurrent threads. Although compilers and hardware take care that the optimization effects are not externally visible for DRF programs, these optimizations can exhibit surprising behavior *in the presence* of data races. For example, a compiler can generate an optimized program that looks like either Figure 2.2(a) or 2.2(b). In the absence of data dependences, it is perfectly legal for current compilers and hardware to reorder the two instructions in Thread 1 (Figure 2.2(a)). This can lead to a null dereference of x. The compiler can also decide to optimize the redundant loads of done from the while loop as shown in Figure 2.2(b), thereby leading to an infinite loop. Note that both these optimizations are perfectly legal for DRF programs.

Data races directly or indirectly lead to concurrency bugs, including sequential consistency, atomicity, and order violations [112, 139] that may corrupt data, cause a crash, or prevent forward progress [69, 98, 133]. The Therac-25 disaster [107], the Northeastern electricity blackout of 2003 [179], and the mismatched NASDAQ Facebook share prices [145] were all due to race conditions, and are a testament to the danger posed by data races.

2.1.2 Detecting Data Races

There has been much work on detecting data races [34, 56, 57, 59, 60, 66, 68, 94, 99, 115, 120, 131, 132, 136, 146, 147, 170, 194]. This section overviews prior work.

2.1.2.1 Static Analyses

Static program analysis for detecting data races has the advantage of being sound by considering all possible inputs, environments, and thread interleavings—but at the cost of precision and scalability [59, 131, 132, 147, 183]. Static analysis abstracts data and control flow conservatively, leading to imprecision and false positives. Achieving soundness, precision and scalability are often conflicting properties—imprecision and performance tend to scale poorly with increasing program size and complexity. For example, *conditional must-not alias analysis* is not so imprecise as previous analyses, but it does not scale to large programs [131]. Sound static analysis is useful for identifying potential data races, but as we and others find [19, 49, 57, 99, 106, 182], dynamic analysis is essential for pruning false positives.

2.1.2.2 Software-Based Dynamic Analysis

Happens-before analysis. Dynamic *happens-before* analysis soundly and precisely checks that all conflicting accesses are ordered by the happens-before relationship [34, 68, 99, 104,

120, 146]. Analyses typically track happens-before using vector clocks [126]; each vector clock operation takes time proportional to the number of threads. In addition to tracking happens-before, an analysis must track *when* each thread last wrote and read each shared variable, in order to check that each access *happens after* every earlier conflicting access.

FastTrack is the current state-of-art dynamically sound and precise happens-before-analysis-based data race detector [68]. FastTrack exploits the following insights. (1) In a race-free program, writes to a variable are totally ordered. (2) In a race-free program, upon a write, all previous reads must happen before the write. (3) The analysis must distinguish between multiple concurrent reads since they all potentially race with a subsequent write. *FastTrack* reduces the cost of tracking last accesses, without missing any data races, by tracking a single last writer and, in many cases, a single last reader [68]. This allows FastTrack to be nearly an order of magnitude faster than prior vector-clock-based techniques because instead of $O(n)$ time and space analysis, it replaces *all write and many read vector clocks with a scalar clock* and almost always performs $O(1)$-time analysis on them.

Despite this optimization, FastTrack still slows executions by nearly an order of magnitude on average (e.g., $>$ 8X slowdown [68]). Its high run-time overhead is largely due to the cost of tracking shared variable accesses, especially reads. A program's threads may perform reads concurrently, but FastTrack requires each thread to update shared metadata on each read. These updates effectively convert the reads into writes that cause remote cache misses [20]. Moreover, FastTrack must synchronize to ensure that its happens-before checks and metadata updates happen atomically. These per-read costs fundamentally limit FastTrack and related analyses (e.g., [73]). Even industry-standard tools such as Intel's

Inspector XE,[1] Google's ThreadSanitizer v2 [162],[2] and Helgrind [134],[3] which are largely based on happens-before analysis, incur substantial slowdowns, e.g., 10–100X. Furthermore, happens-before analysis scales poorly: each vector clock operation takes linear time in terms of the number of threads ($O(n)$ for n threads).

Sampling-based approaches. With a view to minimize performance overhead, researchers have distributed the work of happens-before analysis across many production runs [34, 99, 120]. These techniques give up soundness entirely, missing data races in exchange for performance, usually in order to target production systems. Sampling-based analyses perform analysis selectively at accesses and synchronization operations [34, 120]. However, sampling-based race detection approaches continue to suffer from run-time overhead that is still high, unscalable, and unbounded. *LiteRace* and *Pacer* sample race detection analysis but instrument all program accesses, adding high baseline overhead even when the sampling rate is miniscule [34, 120]. *RaceMob* "crowdsources" race detection analysis by trying to detect just one potential race in each execution [99]. These approaches still incur high overheads and do not scale well due to the high cost of tracking the happens-before relationship [19]. Furthermore, they provide limited coverage guarantees [60, 120].

Happens-before analysis is sound and precise but only reports data races that manifest in the current execution. In other words, coverage is quite *sensitive* to thread interleavings [196]; unrelated happens-before edges mask data races that can manifest in another execution. Since repeated executions in similar environments tend to exercise similar interleavings [143, 160], happens-before analysis is unlikely to detect races exposed by rare interleavings.

[1] `https://software.intel.com/en-us/intel-inspector-xe`

[2] `https://code.google.com/p/thread-sanitizer`

[3] `http://valgrind.org`

Dynamic lockset analysis. Detecting data races during testing runs is insufficient to find all data races, whose occurrence is sensitive to inputs, environments, and thread interleavings [61, 80, 99, 170]. Researchers have considered alternatives in an effort to improve performance and/or data race coverage. An alternative is *lockset* analysis, which detects violations of a locking discipline (e.g., [49, 158, 181]). Dynamic lockset analysis has good coverage but at the cost of precision and performance [158, 181]. To minimize time and space costs, most lockset analyses compute the lockset for each shared variable on the fly, reporting false positives that are not even violations of the locking discipline. The fact that lockset analysis reports false data races limits its value for race detection and so it is unsuitable for providing a strong execution model. In an effort to limit these false positives, these analyses introduce *unsound* heuristics [135, 158, 181]. Alternatively, a more heavyweight analysis can soundly and precisely check the lockset property [49, 136]. Hybrids of happens-before and lockset analysis tend to report false positives (e.g., [136]). Hybrid approaches have tried to effectively combine lockset and happens-before tracking to minimize false positives and performance overheads [136].

Other approaches. *Goldilocks* [57] detects races soundly and precisely and provides exceptional, fail-stop data race semantics. The Goldilocks paper reports 2X average slowdowns, but the authors of FastTrack argue that a realistic implementation would incur an estimated 25X average slowdown [68].

In work concurrent with Valor [20] (Chapter 3), *Clean* detects write–write and write–read data races but does not detect read–write races [159]. Clean exploits the insight that detecting read–write conflicts eagerly is expensive, and does not detect read–write conflicts at all. Instead, Clean argues that detecting write–write and write–read data races is sufficient

to provide certain execution guarantees such as freedom from out-of-thin-air values [118]. Ignoring read–write races allows Clean to provide a weaker memory model than SFR serializability, and still requires extending cache coherence [159].

2.1.2.3 Hardware Support

Custom hardware can accelerate data race detection by adding on-chip memory for tracking vector clocks or locksets and extending cache coherence to identify shared accesses [6, 54, 130, 159, 188, 196], but they require invasive architecture changes that are not applicable to today's systems and manufacturers have been reluctant to change already-complex cache and memory subsystems substantially to support race detection. Furthermore, realistically bounded hardware resources cannot efficiently detect data races that occur over very long spans of dynamic instructions [6, 54, 130].

2.1.2.4 Model Checking

Model checking can detect data races or other concurrency errors by exploring many thread interleavings and/or inputs exhaustively, but it suffers from state-space explosion. *CHESS* reduces state-space explosion somewhat by bounding preemptions and considering one input at a time [129], although scalability is still a challenge especially for long-running programs.

2.1.2.5 Languages and Types

New languages can eliminate data races, but they require writing programs in these potentially restrictive languages [28, 152]. Type systems can ensure data race freedom, but they typically require adding annotations and modifying code [1, 36]; writing annotations is tedious and constraining.

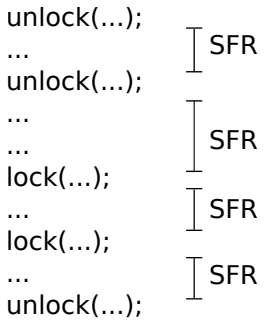


Figure 2.3: Synchronization-free regions (SFRs).

2.1.2.6 Exposing Effects of Data Races

Prior work that is orthogonal to detecting data races exposes erroneous behavior due to data races, often under non-sequentially-consistent memory models. Prior work tries to infer which data races are most likely to be harmful (e.g., crash the program, hurt performance, or corrupt data) [37, 69, 98, 133], and several approaches try to expose errors by forcing an execution to perform rare but allowable behavior at racy accesses [37, 69, 133, 160]. We and other researchers note that every data race is potentially harmful because language memory models such as C/C++ and Java provide no or very weak semantics for data races [2, 30, 133] (Section 2.2.1).

2.1.2.7 Tolerating Data Races

ToleRace [151] and ISOLATOR [149] detect and tolerate some concurrency errors including "asymmetric" data races, in which one thread follows a locking discipline but another does not. These techniques work by making local copies of shared accesses in critical sections, and detect and tolerate conflicts when critical sections end.

BulkCompiler and *Atom-Aid* rely on custom hardware support to provide bulk-atomic execution of dynamic regions, tolerating some effects of data races [7, 116]. Software-based approaches can provider stronger behavioral guarantees for racy executions, but at additional cost [139, 161].

2.1.2.8 Detecting Region Conflicts

Given the high cost of sound and precise happens-before data race detection, prior work has sought to detect the subset of data races that may violate serializability of an execution's synchronization-free regions (SFR). An SFR is a sequence of dynamic instructions separated by synchronization operations (e.g., acquire/release, fork/join, wait/notify). Figure 2.3 shows an example of SFRs. Every executed non-synchronization instruction is in exactly one SFR.

Several techniques detect conflicts between operations in SFRs that overlap in time [56, 115, 121]. SFR conflict detection yields the following guarantees: a DRF execution produces no conflicts; any conflict is a data race; and a conflict-free execution is equivalent to a serialization of SFRs.

Prior work called *Conflict Exceptions* (CE) detects conflicts between overlapping SFRs and treats conflicts as exceptions [115]. CE achieves high performance via hardware support for conflict detection that augments existing cache coherence mechanisms. However, its hardware support has several drawbacks. First, it adds complexity to the cache coherence mechanism. Second, each cache line incurs a high space overhead for storing metadata. Third, sending larger coherence messages that include metadata leads to coherence network congestion and requires more bandwidth. Fourth, cache evictions and synchronization operations for regions with evictions become more expensive because of the need to preserve

metadata by moving it to and from memory. Fifth, requiring new hardware precludes use in today's systems.

DRFx detects conflicts between regions that are synchronization free but also *bounded*, i.e., every region has a bounded maximum number of instructions that it may execute [121]. Bounded regions allow DRFx to use simpler hardware than CE [121, 167]. *DRFx* detects conflicts among bounded regions by maintaining region buffers and Bloom filter signatures of memory accesses [121, 167]. Detecting conflicts among bounded regions means DRFx cannot detect all violations of SFR serializability, although it guarantees SC for conflict-free executions. Furthermore, DRFx broadcasts the Bloom filter signatures and occasionally the region buffers across cores, which is unscalable for large regions (e.g., SFRs) and with increasing core counts. Like CE, DRFx is inapplicable to today's systems because it requires hardware changes.

IFRit detects data races by detecting conflicts between dynamically overlapping *interference-free regions* (*IFRs*) [56]. An IFR is a region of one thread's execution that is associated with a particular variable, during which another thread's read and/or write to that variable is a data race. IFRit comprises both static and dynamic analysis. IFRit relies on whole-program static analysis to place IFR boundaries conservatively, so IFRit is precise (i.e., no false positives). Conservatism in placing boundaries at data-dependent branches, external functions calls, and other points causes IFRit to miss some IFR conflicts that may compromise SFR serializability [56]. This leads IFRit to focus on precisely detecting as many races as possible, but it adds high run-time overhead.

2.1.2.9 Summary

Despite much effort, data races remain a challenge. Their occurrence is environment, input, and timing sensitive [61, 80, 99, 170]. Moreover, programmers introduce

data races both accidentally (e.g., when optimizing contention), and/or intentionally (for performance) [69, 98, 133]. Unfortunately, existing sound and precise data race and region conflict detectors are impractical, relying on custom hardware or slowing programs substantially [56, 68, 115, 167].

2.2 Memory Models

A memory consistency model (or simply a memory model) is an interface between the software and the hardware, and defines the set of possible orders in which memory operations can interleave and the possible values returned by a read [2, 172]. A memory model is important since it specifies the allowed behaviors of a multithreaded program executing under the shared memory programming model. According to Sorin et al. [172], a good memory consistency model must possess the following properties:

1. PROGRAMMABILITY: A good model should be intuitive to use and should make multithreaded programming (relatively) easy.

2. PERFORMANCE: A good model should allow high-performance implementations at reasonable power and cost.

3. PORTABILITY: A good model should be flexible enough to be adopted widely and optionally provide the ability to translate among models.

4. PRECISION: A good model should be formally defined and unambiguous.

Pervasive shared-memory programming languages (e.g., C/C++ and Java) provide weak memory models. Strong memory models exist, but are limited by cost, complexity, and semantics. Here we discuss the landscape and reveal the critical gap that this dissertation fills.

2.2.1 Weak Memory Models

Current shared-memory programming language and hardware memory models are *weak* since they provide no or poor guarantees for racy executions.

2.2.1.1 DRF0-Based Language Memory Models

DRF0 (data-race-free-0) is a memory model introduced by Adve and Hill in 1990 [5]. The motivation behind DRF0 is that consistency models weaker than sequential consistency [105] (SC) (Section 2.2.2.1) are hard for programmers to understand. Thus, DRF0 guarantees SC for DRF executions. The rationale for the DRF0 memory model is that it permits compilers and hardware to perform aggressive intra-thread optimizations, as long as they do not arbitrarily reorder memory accesses across synchronization operations. For DRF programs, the effects of optimizations will not be externally visible.

Current high-level programming languages such as C/C++ and Java extend the DRF0 memory model [31,118]. DRF0 (and its variants) provide SC for well-synchronized, or DRF executions. In addition to SC, DRF0 (and its variants) provide an even stronger guarantee than SC but *only* for DRF programs: serializability of SFRs [2,31,115,118] (Section 2.2.2.2).

However, *for executions with data races, DRF0 provides no useful guarantees.* In C/C++, data races have undefined semantics [2, 31] creating a problem, not just in theory, but in practice: a recent study showed that C/C++ programs with seemingly "benign" data races may behave incorrectly due to compiler transformations or microarchitectural changes not visible to the programmer [29]. In contrast, Java's memory model preserves memory and type safety despite races, but permits non-SC behaviors [118]. Unfortunately, Java's safety guarantees preclude important compiler optimizations [33, 163], so current Java virtual machines (JVMs) do *not* actually enforce Java's memory model [33].

2.2.1.2 Hardware Memory Models

Relaxed consistency. Relaxed consistency is a class of hardware memory models that require no ordering constraints between two memory operations from the same thread for all four combinations of load (memory read) and store (memory write) operations: Load $\rightarrow$ Load, Load $\rightarrow$ Store, Store $\rightarrow$ Store, and Store $\rightarrow$ Load. Relaxed or weak memory consistency models only preserve the orderings that are *explicitly required* by a programmer [172]. The motivation behind this memory model is to allow more reorderings by the compiler and the hardware to improve performance. Some examples of relaxed consistency memory models are release consistency [79], IBM Power [157], and ARM [8]. However, using relaxed consistency memory models is complex since programmers must understand low-level reordering issues to know where to "order" instructions. Moreover, vendors have failed to agree on a single relaxed memory consistency model, compromising portability [172].

Total store order. Total store order (TSO) is a memory model that allows Store $\rightarrow$ Load reorderings, to hide the latency of writes. Note that TSO preserves the other three constraints: Load $\rightarrow$ Load, Load $\rightarrow$ Store, and Store $\rightarrow$ Store. TSO and its variants are provided by widely-used architectures such as x86 and SPARC [164, 174]. TSO is primarily an axiomatic description of the possible behaviors in a multicore processor with write buffers.

2.2.2 Strong Memory Models

Researchers have argued that systems must provide stronger memory models to avoid unusably complex semantics for data races [2, 41]. Adve and Boehm emphasize that systems

should give simple, well-defined semantics to executions with data races using hardware and software support [2]. According to Adve and Boehm [2],

> *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.*
>
> ...
>
> *This process has exposed fundamental shortcomings in our languages and a hardware–software mismatch. Semantics for programs that contain data races seem fundamentally difficult, but are necessary for concurrency safety and debuggability. We call upon software and hardware communities to develop languages and systems that enforce data-race-freedom, and co-designed hardware that exploits and supports such semantics.*

Furthermore, providing such strong semantics to executions both with or without data races would help simplify the programming language semantics [20, 115].

2.2.2.1 Sequential Consistency

Sequential consistency (SC) [105] is a memory model that requires all memory operations in an execution to appear to have executed in a global sequential order that is consistent with the per-thread program order. This memory model is simple, as it matches the operational intuition where each read from a location sees the value of the latest write [172]. SC preserves the order of two memory operations from the same thread for all four combinations of load and store operations. However, many program transformations that are sequentially valid can potentially violate SC in the presence of multiple threads, such as, reordering of instructions in one thread can be seen by another thread as in Figure 2.2(a). As a result, SC precludes the use of common compiler optimizations (code motion, loop transformations, etc.) and hardware optimizations (out-of-order execution, store buffers, lockup-free caches, etc.).

Much work has focused on providing SC as a memory model [4, 78, 109, 110, 122, 150, 165, 168, 178] either end-to-end (enforced relative to the original program) or for the compiled program only. For example, hardware-based SC enforcement provides SC for the compiled program only. However, as discussed, providing end-to-end SC requires corresponding restrictions of reordering in the compiler in addition to the hardware, which slows programs and/or relies on custom hardware support [4, 78, 122, 168].

Still, SC is not a particularly strong memory model, and prior work argues that "programmers do not reason about correctness of parallel code in terms of interleavings of individual memory accesses, and sequential consistency does not prevent common sources of concurrency bugs..." [2]. Some operations that many programmers expect to execute atomically do not execute atomically under SC (e.g., multi-access operations such as x++ and buffer[index++] = ...). Furthermore, it is challenging for program analyses and runtime systems to deal with all interleavings at the granularity of individual memory accesses.

2.2.2.2 SFR Serializability

An execution is region serializable if it is equivalent to some serial execution of regions (i.e., some global order of non-interleaved regions). In the *SFR serializability* memory model, all regions of code between synchronization operations appear to execute atomically and in program order, even if they have data races. This behavior is appealing because it *extends the same guarantees to executions with races that DRF0 gives to DRF executions* (Portability, Programmability). Another advantage of SFR serializability is that compiler and hardware optimizations are free to reorder memory accesses within SFRs (Performance). Furthermore, *architecture support alone* (e.g., [115]) can ensure *end-to-end* SFR serializability because compilers already respect synchronization boundaries (Portability). SFR serializability is a

stronger alternative memory model compared to DRF0 and SC since it ensures serializability of SFRs even for executions with data races.

Enforcing region serializability seems inherently problematic and expensive due to the need to support unbounded speculative execution. Moreover, trying to ensure region serializability can lead to deadlocks or livelocks for racy executions thereby complicating the semantics. Researchers have thus introduced a memory consistency model that treats data races as errors, potentially throwing a consistency exception for a data race, but otherwise ensuring region serializability [20, 115]. (Furthermore, other approaches also treat some or all data races as errors [57, 159, 188].)

Data race exceptions. Instead of ensuring serializability, a cleaner way to deal with data races is to detect *problematic* data races and halt the execution with a *data race exception* when one occurs [57, 115, 121]. Problematic data races are ones that may violate a useful semantic property, like SC [57] or SFR serializability [115]. Exceptional *fail-stop* semantics for data races have several desirable benefits [115]. First, they simplify programming language specifications because data races have clear semantics, and an execution is either correct or it throws an exception. Second, they make software safer by limiting the possible effects of data races. Third, they permit aggressive optimizations within SFRs that might introduce incorrect behavior with weaker data race semantics. Fourth, they help debug problematic data races by making them a fail-stop condition. Treating problematic data races as exceptions requires a mechanism that dynamically detects those races precisely (no false positives) and is efficient enough for always-on use.

Prior work called Conflict Exceptions [115] avoids the expense of detecting all happens-before races by instead detecting conflicts between SFRs. Every SFR conflict is a true data

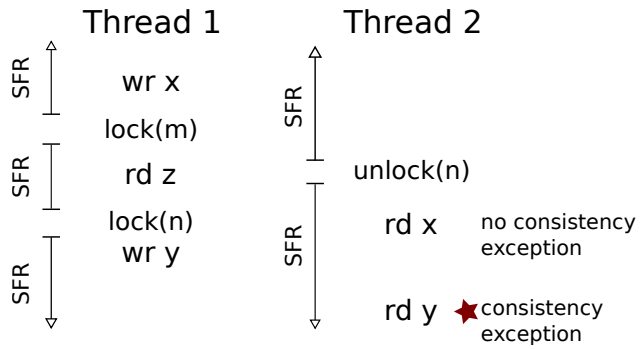


Figure 2.4: Under SFRSx, an execution generates an exception *only* for a data race that may violate SFR serializability.

race, but not every data race is a conflict. Conflict Exceptions (CE) executes a program and either reports an exception on a SFR conflict or ensures serializability of SFRs. We call this memory model *SFRSx*. SFRSx guarantees (1) SFR serializability for DRF executions and (2) an exception for any non-SFR-serializable execution. Under SFRSx, an execution with a data race may either enforce SFR serializability or generate an exception. Figure 2.4 shows an execution with data races on two shared variables, x and y. An implementation of SFRSx does *not* generate a consistency exception at the read of x because the SFRs accessing x do not overlap. In contrast, SFRSx generates an exception at the read of y (because the SFRs accessing y overlap), to avoid violating SFR serializability (e.g., suppose Thread 1's SFR later writes x or y).

CE provide SFRSx, but with significant drawbacks. CE adds hardware on top of an already-complex MOESI cache coherence protocol to detect SFR conflicts [115]. CE generates prohibitively high memory traffic (Section 4.6.2.3) and fundamentally relies on a cache coherence mechanism that uses messages to eagerly update coherence permissions, to

exchange access metadata. CE thus requires invasive hardware customizations that makes the design unscalable and unimplementable.

2.3 Checking Atomicity of Programmer-Defined Regions

SFR serializability ensures serializability of regions demarcated by synchronization operations. A higher-level semantic property of a program is serializability of programmer-defined regions that are meant to be *atomic*. Such atomic regions are also referred to as transactions. While checking for region conflicts suffice to provide SFR serializability, it is important to check serializability of atomic regions rather than conflict freedom since region conflicts are not necessarily errors according to the program semantics. In the following, we briefly discuss about atomicity and the related work.

2.3.1 Overview

Atomicity is a fundamental non-interference property that eases reasoning about program behavior in multithreaded programs. For programming language semantics, atomicity is synonymous with *serializability*: program execution must be equivalent to some serial execution of atomic arbitrarily-sized regions and non-atomic instructions. That is, the code block's execution *appears* not to be interleaved with statements from other concurrently executing threads. This effectively means that the execution should be equivalent to one for which *intermediate updates* from within an atomic section are not externally visible until the execution of the atomic section is complete. Programmers can thus reason about atomic regions without considering effects of other threads. In some sense, atomicity can be considered to be consistency of the program at a semantic level.

While much work has considered how to detect and prevent data races (Section 2.1), ensuring data-race-freedom does not guarantee semantic correctness—in particular, it does

not guarantee atomicity of regions that the programmer intended to be atomic. Except for the fact that data races are often indicators of atomicity violations, they are complementary errors [73, 84, 113]. Consider the following example from prior work [73]:

```
class Set {
  final Vector elems = new Vector();
  atomic void add(Object x) { // intended to be atomic
    if (!elems.contains(x)) {
      elems.add(x);
    }
  }
}


class Vector {
  synchronized void add(Object o) { ... }
  synchronized boolean contains(Object o) { ... }
}
```

Note that the atomic keyword is used just to indicate the programmer's *intention* that the program will always execute Set.add() atomically. The atomic keyword has no effect on program semantics (e.g., the language and runtime system do not enforce atomicity based on the atomic keyword).

The method Set.add() is data race free: the methods called from Set.add() synchronize on the this object.[4] However, it is possible for Set.add() to violate atomicity: if two threads call Set.add(x) with parameters referencing the same object, it is possible for the referenced object to be added to the set twice—an *unserializable* outcome (i.e., an outcome impossible if atomic regions are serialized).

Atomicity specification. An *atomicity specification* denotes particular code regions that are expected to execute atomically. Program analysis can check atomicity by checking whether a program conforms to the atomicity specification. A *violation* indicates that the

[4] A synchronized method executes its method body in a critical section that acquires the lock of this object.

program or specification is wrong (or both). Writing an atomicity specification may seem burdensome, but prior work shows that specifications can be derived mostly automatically [67, 73].

Prior work finds that a reasonable *starting point* for an atomicity specification is to assume that all methods and/or synchronization blocks are atomic [67, 72, 73]. From there, programmers can iteratively apply an atomicity-checking analysis to refine the specification. In particular, programmers investigate each reported atomicity violation and decide whether (1) it is a true atomicity violation, which needs to be fixed; or (2) the offending method or synchronization block is not intended to be atomic, in which case the atomicity specification needs to be refined.

2.3.2 Detecting Atomicity Violations

This section describes static and dynamic analyses related to detecting atomicity violations, including the current state-of-art called Velodrome [73].

Static analyses. Similar to detecting data races, static approaches can check all inputs soundly, but they are imprecise, and in practice they do not scale well to large programs nor to dynamic language features such as dynamic class loading. Type systems can help check atomicity but require a combination of type inference and programmer annotations [72, 74]. Model checking does not scale well to large programs because of state space explosion [62, 65, 90]. Static approaches are well suited to analyzing critical sections but not wait–notify synchronization.

Checking conflict serializability. Checking serializability efficiently is impractical [140], so existing analyses check properties sufficient for serializability such as *conflict serializability*. An execution is conflict serializable if and only if the graph of dependences among executing transactions[5] is acyclic.

Velodrome is a state-of-the-art dynamic analysis that checks conflict serializability soundly and precisely [73]. Each code region that is supposed to execute atomically (according to the atomicity specification) executes as a *transaction*. Other accesses each execute as a *unary transaction*. Velodrome's dynamic analysis builds a graph of transactions at run time. When a new (regular or unary) transaction starts, the analysis adds an intra-thread dependence edge from the thread's prior transaction to the new transaction. At each access, the analysis detects cross-thread data dependences: write–read, read–write, and write–write dependences between threads, as well as release–acquire synchronization dependences. Velodrome adds cross-thread dependence edges between transactions as the program executes. It detects cycles in the graph; a cycle is a sound and precise condition for a conflict serializability violation.

Velodrome slows programs by 12.7X in prior work [73] and 6.1X using our Velodrome implementation and experiments (Section 5.6.3). These slowdowns are largely due to tracking cross-thread dependences soundly and precisely, which has two main costs. First, tracking dependences involves maintaining the last transaction to write, and each thread's last transaction to read, each variable. Second, to preserve correctness in the face of accesses potentially involved in data races, the analysis must use atomic operations and memory fences to ensure that an access and its corresponding analysis execute together atomically. Atomic operations and memory fences slow execution by limiting reordering and serializing

[5] A transaction represents a dynamically executing atomic region. Each non-atomic access executes as a unary transaction.

in-flight instructions and by triggering remote cache misses. It is especially expensive for shared, mostly read-only variables, since metadata updates lead to remote cache misses that could otherwise be avoided.

Farzan and Parthasarathy introduce a dynamic serializability-checking analysis based on finding cycles among transactions [63]. Their analysis bounds space overhead optimally so space is *not* proportional to the length of the run, by summarizing the dependence graph as transactions finish.

While Velodrome detects cycles online as the program executes, Farzan and Parthasarathy's analysis detects cycles offline, i.e., after the execution finishes. They compare their summarized dependence graph to an *unsummarized* dependence graph—but this unsummarized graph does not use garbage collection (GC), so its space overhead is unavoidably proportional to the length of the run.

Other dynamic analyses. Some dynamic approaches are *predictive*, so they detect atomicity violations not only for the current execution's thread interleavings, but also for other interleavings that could have executed [169, 173]. Predictive analyses tend to be considerably more expensive than non-predictive analysis, particularly as they aim to provide higher coverage and less imprecision.

Wang and Stoller propose dynamic analyses for checking atomicity based on detecting unserializable patterns [184, 185]. These approaches are predictive since they aim to find potential violations in other executions, but this process is inherently imprecise, so they may report false positives.

Atomizer is a dynamic atomicity checker that uses a variation of the lockset algorithm [158] to determine shared variables that can have racy accesses, and monitors those

variables for potential atomicity violations. Atomizer reports false positives since it relies on the locket algorithm.

Other alternatives. *HAVE* combines static and dynamic analysis to obtain benefits of both approaches [45]. Because HAVE speculates about untaken branches, it can report false positives.

Several approaches infer an atomicity specification automatically [47, 84, 113, 175, 189], which is useful because specifications are not usually available. However, these approaches are inherently unsound and imprecise. Furthermore, many of these approaches add high overhead to track cross-thread dependences, e.g., AVIO slows programs by more than an order of magnitude [113].

Prior work *exposes* atomicity violations by making them more likely to occur [142, 143] and thus more likely for a non-predictive analysis to detect. Exposing atomicity violations is complementary to checking atomicity.

Transactional memory enforces programmer-specified atomicity annotations by speculatively executing atomic regions as *transactions*, which are rolled back if a region conflict occurs [91]. Custom hardware–based TM approaches offer low overhead [128], but any real hardware support is likely to be limited and require software TM. Software TM systems (STM) suffer from two main problems: poor performance and weak semantics [39]. Supporting *strong atomicity* semantics using STM affects the performance even more.

Atom-Aid relies on custom hardware to execute regions atomically and to detect some atomicity violations [116]. Static analysis can infer needed locks automatically from an atomicity specification (e.g., [46]). The inferred locks are inherently imprecise, leading to overly conservative locking.

Chapter 3: Valor: Providing RSx Efficiently in Software

3.1 Introduction

Memory models of shared-memory programming languages such as C/C++ and Java provide virtually no guarantees for executions with data races [2, 31, 118]. Chapters 1 and 2 highlight the the complexity and risk associated with data races. Such poor (zero or weak) semantics for racy program executions is unsatisfactory, especially in the context of safety-critical applications. We and other researchers have argued that this lack of semantic guarantees create an urgent need to strengthen memory models [2, 30].

The key to dealing with the consequences of data races, regardless of the memory consistency model, is providing an efficient mechanism for identifying the *conflicting* memory accesses that make up a data race. Efficient system support for *conflict detection* is essential to comprehensible language specifications [2, 20, 42, 115, 121, 159, 167] and simple execution models [85, 86, 88], as well as system support such as debugging [56, 73] and determinism [13, 53, 93, 180].

SFRSx is a desirable strong memory model that provides several benefits to ensure strong semantics for *all* program executions, such as simplifying programming and debugging, and allowing flexibility in programming language implementations, including the freedom to rearrange operations in the compiler and architecture [2, 20, 41, 139]. To that extent,

this chapter focuses on a *novel, software-only solution* to provide SFRSx to all program executions.

Motivation. One way to deal with data races is to provide exceptional, fail-stop semantics, which have several desirable benefits [115] (Section 2.2.2). The secondary motivation for our work is that *without any existing, fully precise, high-performance software data race detector or region conflict detector* (Section 2.1), race detection is largely limited to debugging and in-house testing [120]. An efficient, precise data race detector is thus invaluable for finding and fixing data races, both before and after deployment. During development, high overheads are problematic, wasting limited resources [81]. Developers shy away from using intrusive tools that do not allow them to test realistic program executions [120]. Detecting data races during testing and debugging is an essential step in producing reliable software and doing so presents several important challenges. Moreover, the manifestation of a data race is dependent on an execution's inputs, environments, and thread interleavings, and data races in deployed systems may cause yet-unseen failures, even in thoroughly tested programs [107, 145, 179]. A data race may not occur in hours of program execution [196], sometimes requiring weeks to reproduce, diagnose, and fix if it is contingent on specific environmental conditions [80, 112, 179]. Detecting and debugging such data races requires analyzing production executions, making performance a key constraint. *In this work, we develop a mechanism that meets this requirement by detecting problematic races efficiently and precisely.*

Existing work. Sound and precise dynamic data race detection enables a language to define the semantics of data races by throwing an exception for every data race. Section 2.1 discusses existing work related to detecting data races. The precision and scalability

limitations make static approaches unsuitable for detecting data races and providing a strong execution model. Unfortunately, existing sound and precise data race and region conflict detectors are impractical, relying on custom hardware or slowing programs substantially [56, 115, 167].

The next section introduces our software-only analyses (like IFRit [56]) that detect conflicts between SFRs (like CE [115]).

3.2 Efficient Region Conflict Detection

The goal of this work is to develop a region conflict detection mechanism that is useful for providing guarantees to a programming language implementation, and is efficient enough for always-on use. The rest of this chapter uses "region" and "SFR" interchangeably. In addition to SFRs, we show that our proposed region conflict detection mechanism is also applicable to regions that are only demarcated by synchronization "release" operations (Section 3.5.1).

We explore two different approaches for detecting region conflicts. The first approach is called *FastRCD*, which, like FastTrack [68], uses epoch optimizations and *eagerly* detects conflicts at conflicting accesses. We have developed FastRCD in order to better understand the characteristics and performance of a region conflict detector based on FastTrack's approach. Despite FastRCD being a natural extension of the fastest known sound and precise dynamic data race detection analysis, Section 3.9.2 experimentally shows that FastRCD's need to track last readers imposes overheads that are similar to FastTrack's and are too high for always-on use.

In response to FastRCD's high overhead, we develop ***Valor***,[6] *which is the main contribution of this chapter*. Valor detects write–write and write–read conflicts *eagerly* as in FastRCD. The key to Valor is that it detects read–write conflicts *lazily*, effectively avoiding the high cost of tracking last-reader information. Lazy conflict detection is more efficient than eager conflict detection because it need not track last reader information. Instead, in Valor, each thread logs read operations locally. At the end of a region, the thread *validates* its read log, checking for read–write conflicts between its reads and any writes in other threads' ongoing regions. By *lazily* checking for these conflicts, Valor can provide fail-stop semantics without hardware support and with overheads far lower than even our optimized FastRCD implementation.

Section 3.3 describes the details of FastRCD and the fundamental sources of high overhead that eager conflict detection imposes. Section 3.4 then describes Valor and the implications of lazy conflict detection. Sections 3.5 and 3.6 describe extensions and optimizations for FastRCD and Valor. We prove that Valor is a sound and precise region conflict detector in Section 3.7. We discuss implementation details in Section 3.8 and present results in Section 3.9. We conclude in Section 3.10.

3.3 FastRCD: Detecting Conflicts Eagerly in Software

This section presents FastRCD, a new software-only dynamic analysis for eagerly detecting region conflicts. FastRCD reports a conflict when a memory access executed by one thread conflicts with a memory access that was executed by another thread in a region that is *ongoing*. It provides essentially the same semantics as CE [115] but *without* hardware support.

[6]Valor is an acronym for Validating anti-dependences lazily on release.

In FastRCD, each thread keeps track of a clock c that starts at 0 and is incremented at every region boundary. This clock is analogous to the logical clocks maintained by FastTrack to track the happens-before relation [68, 104].

FastRCD uses *epoch optimizations* based on FastTrack's optimizations [68] (Section 2.1.2.2) for efficiently tracking read and write metadata. It keeps track of the single last region to write each shared variable, and the last region or regions to read each shared variable. To do so, FastRCD performs the following steps. For each shared variable x, FastRCD maintains x's last writer region using an epoch c@t: the thread t and clock c that last wrote to x. When x has no concurrent reads from overlapping regions, FastRCD represents the last reader as an epoch c@t. Otherwise, since there is no ordering among concurrent readers, FastRCD keeps track of last readers in the form of a *read map*, containing one entry per thread that maps threads to the clock values c of their last read to x. We use the following notations to help with exposition:

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

epoch(T) – Returns an epoch c@T, where c represents the ongoing region in thread T.

$\mathcal{W}_x$ – Represents last writer information for variable x in the form of an epoch c@t.

$\mathcal{R}_x$ – Represents a read map for variable x of entries t $\rightarrow$ c. $\mathcal{R}_x$[T] returns the clock value c when T last read x (or 0 if not present in the read map).

Our algorithms use T for the current thread, and t and t' for other threads. For clarity, we use a common notation for read epochs and read maps; a one-entry read map is a read epoch, and an empty read map is the initial-state epoch 0@0.

Algorithms 1 and 2 show FastRCD's analysis at program writes and reads, respectively. At a write by thread T to program variable x, the analysis first checks if the last writer epoch

matches the current epoch, indicating an earlier write in the same region, in which case the analysis does nothing (line 1). Otherwise, it checks for conflicts with the previous write (lines 3–4) and reads (lines 6–9). Finally, it updates the metadata to reflect the current write (lines 11–12).

Algorithm 1 WRITE [FastRCD]: thread T writes variable x

```
1:  if W_x ≠ epoch(T) then                          ▷ Write in same region
2:      let c@t ← W_x
3:      if c = clock(t) then                        ▷ t's region is ongoing
4:          Conflict!                               ▷ Write–write conflict detected
5:      end if
6:      for all t' → c' ∈ R_x do
7:          if c' = clock(t') then
8:              Conflict!                           ▷ Read–write conflict detected
9:          end if
10:     end for
11:     W_x ← epoch(T)                              ▷ Update write metadata
12:     R_x ← ∅                                     ▷ Clear read metadata
13: end if
```

At a read, the instrumentation first checks for an earlier read in the same region, in which case the analysis does nothing (line 1). Otherwise, it checks for a conflict with a prior write by checking if the last writer thread t is still executing its region c (lines 3–5). Finally, the instrumentation updates T's clock in the read map (line 6).

Algorithm 2 READ [FastRCD]: thread T reads variable x

```
1:  if R_x[T] ≠ clock(T) then                       ▷ Read in same region
2:      let c@t ← W_x
3:      if c = clock(t) then                        ▷ t's region is ongoing
4:          Conflict!                               ▷ Write–read conflict detected
5:      end if
6:      R_x[T] ← clock(T)                           ▷ Update read map
7:  end if
```

FastRCD's analysis at a read or write must execute *atomically*. Whenever the analysis needs to update x's metadata ($\mathcal{W}_x$ and/or $\mathcal{R}_x$), it "locks" x's metadata for the duration of the action (not shown in the algorithms). Because the analyses presented in Algorithms 1 and 2 read and write multiple metadata words, the analyses are not amenable to a "lock-free" approach that updates the metadata using a single atomic operation.[7] Note that the analysis and program memory access need *not* execute together atomically because the analysis need not detect the order in which conflicting accesses occur, just that they conflict.

FastRCD soundly and precisely detects every region conflict just before the conflicting access executes. FastRCD guarantees that region-conflict-free executions are region serializable, and that every region conflict is a data race. It suffers from high overheads (Section 3.9.2) because it unavoidably performs expensive analysis at reads. Multiple threads' concurrent regions commonly read the same shared variable; updating per-variable metadata at program reads leads to communication and synchronization costs *not* incurred by the original program execution.

3.4 Valor: Detecting Read–Write Conflicts Lazily

This section describes the design of *Valor*, a novel, software-only region conflict detector that eliminates the costly analysis on read operations that afflicts FastRCD (and FastTrack). Like FastRCD, Valor reports a conflict when a memory access executed by one thread conflicts with a memory access previously executed by another thread in a region that is ongoing. Valor soundly and precisely detects conflicts that correspond to data races and

[7]Recent Intel processors provide Transactional Synchronization Extensions (TSX) instructions, which support multi-word atomic operations via hardware transactional memory [193]. However, recent work shows that existing TSX implementations incur high per-transaction costs [125, 153].

provides the same semantic guarantees as FastRCD. Valor detects write–read and write–write conflicts exactly as in FastRCD, but detects read–write conflicts differently. Each thread locally logs its current region's reads and detects read–write conflicts *lazily* when the region ends. Valor eliminates the need to track the last reader of each shared variable explicitly, avoiding high overhead.

3.4.1 Overview

During a region's execution, Valor tracks each shared variable's last writer *only*. Last writer tracking is enough to eagerly detect write–write and write–read conflicts. Valor does not track each shared variable's last readers, so it cannot detect a read–write conflict at the conflicting write. Instead, Valor detects a read–write conflict lazily, when the (conflicting read's) region ends.

This section presents Valor so that it tracks each shared variable's last writer using an epoch, just as for FastRCD. Section 3.6 presents an *alternate* design of Valor that represents the last writer differently using *ownership* information (for implementation reasons). Conceptually, the two designs work in the same way, e.g., the examples in this section apply to the design in Section 3.6.

Write–write and write–read conflicts. Figure 3.1(a) shows an example execution with a write–read conflict on the shared variable x. Dashed lines indicate region boundaries, and the labels j-1, j, k-1, etc. indicate threads' clocks, incremented at each region boundary. The grey text above and below each program memory access (e.g., $\langle \mathtt{v}, \mathtt{p@T0} \rangle$) shows x's last writer metadata. Valor stores a tuple $\langle \mathtt{v}, \mathtt{c@t} \rangle$ that consists of a *version*, v, which the analysis increments on a region's first write to x, and the epoch c@t of the last write to x. Valor needs versions to detect conflicts precisely, as we explain shortly.

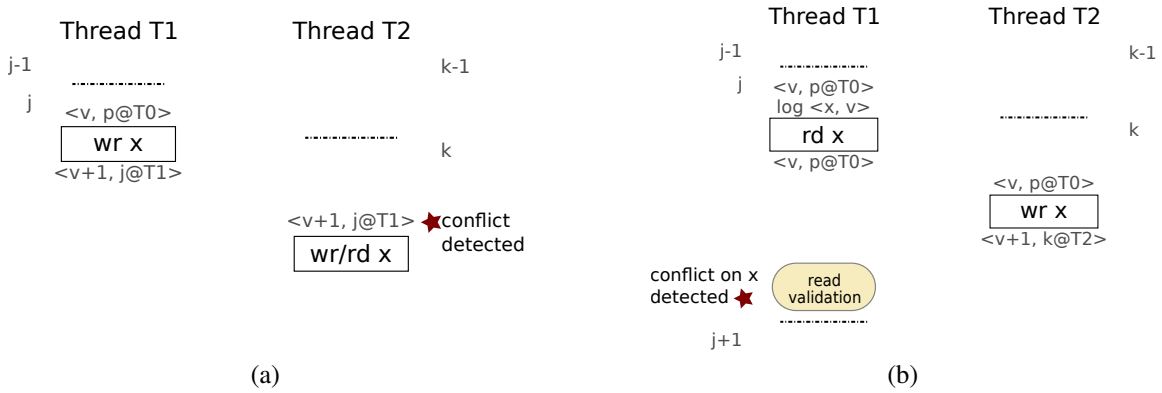


Figure 3.1: (a) Like FastRCD, Valor eagerly detects a conflict at T2's access because the last region to write x is ongoing. (b) *Unlike* FastRCD, Valor detects read–write conflicts lazily. During read validation, T1 detects a write to x since T1's read of x.

In the example, T1's write to x triggers an update of its last writer metadata to $\langle$v+1, j@T1$\rangle$. (The analysis does not detect a write–write conflict at T1's write because the example assumes that T0's region p, which is not shown, has ended.) At T2's write or read to x, the analysis detects that T1's current region is j and that x's last writer epoch is j@T1. These conditions imply that T2's access conflicts with T1's ongoing region, so T2 reports a conflict.

Read–write conflicts. Figure 3.1(b) shows an example read–write conflict. At the read of x, T1 records the read in its thread-local *read log*. A read log entry, $\langle$x, v$\rangle$, consists of the address of variable x and x's current version, v.

T2 then writes x, which is a read–write conflict because T1's region j is ongoing. However, the analysis *cannot* detect the conflict at T2's write, because Valor does not track x's last readers. Instead, the analysis updates the last writer metadata for x, including incrementing its version to v+1.

When T1's region j ends, Valor *validates* j's reads to lazily detect read–write conflicts. Read validation compares each entry $\langle x, v \rangle$ in T1's read log with x's current version. In the example, x's version has changed to v+1, and the analysis detects a read–write conflict. Note that even with lazy read–write conflict detection, Valor guarantees that each conflict-free execution is region serializable. In contrast to eager detection, Valor's lazy detection cannot deliver *precise exceptions*. An exception for a read–write conflict is only raised at the end of the region executing the read, *not* at the conflicting write, which Section 3.4.3 argues is acceptable for providing strong behavior guarantees.

Valor requires versions. Tracking epochs alone is insufficient: Valor's metadata must include versions. Let us assume for exposition's sake that Valor tracked only epochs and not versions, and it recorded epochs instead of versions in read logs, e.g., $\langle x, p@T0 \rangle$ in Figure 3.1(b). In this particular case, Valor without versions correctly detects the read–write conflict in Figure 3.1(b).

However, in the general case, Valor without versions is either unsound or imprecise. Figures 3.2(a) and 3.2(b) illustrate why epochs alone are insufficient. In Figure 3.2(a), no conflict exists. The analysis should *not* report a conflict during read validation, because even though x's epoch has changed from the value recorded in the read log, T1 itself is the last writer.

In Figure 3.2(b), T1 is again the last writer of x, but in this case, T1 *should* report a read–write conflict because of T2's intervening write. (No *write–write* conflict exists: T2's region k ends before T1's write.) However, using epochs alone, Valor cannot differentiate these two cases during read validation. Note that there is no *write–write* conflict because T1's write occurs *after* T2's region k completes.

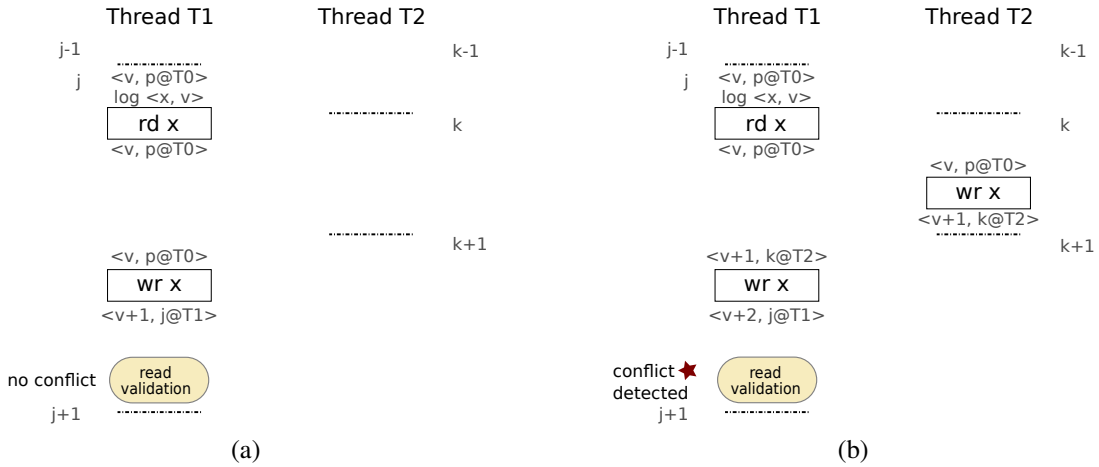


Figure 3.2: Valor relies on versions to detect conflicts soundly and precisely. Associating only epoch information with writes can lead to false positives in inferring read–write region conflicts lazily. (b) Without tracking versions, read validation in thread T1 is unaware of the remote write and cannot detect a read–write conflict lazily.

Thus, Valor uses versions to differentiate cases like Figures 3.2(a) and 3.2(b), in which remote writes may have interleaved between a region's read followed by a write to the same variable. Read validation detects a conflict for x if (1) its version has changed and its last writer thread is not the current thread **or** (2) its version has changed at least *twice*,[8] definitely indicating intervening write(s) by other thread(s).

Read validation using versions detects the read–write conflict in Figure 3.2(b). Although the last writer is the current region (j@T1), the version has changed from v recorded in the read log to v+2, indicating an intervening write from a remote thread. Read validation (correctly) does *not* detect a conflict in Figure 3.2(a) because the last writer is the current region, and the version has only changed from v to v+1.

[8]Note that a region increments x's version only the *first* time it writes to x (line 2 in Algorithm 3).

The rest of this section describes the Valor algorithm in detail: its actions at reads and writes and at region end, and the guarantees it provides.

3.4.2 Analysis Details

Our presentation of Valor uses the following notations, some of which are the same as or similar to FastRCD's notations:

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

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

$\mathcal{W}_x$ – Represents last writer metadata for variable x, as a tuple $\langle \mathsf{v}, \mathsf{c@t} \rangle$ consisting of the version v and epoch c@t.

T.readLog – Represents thread T's read log. The read log contains entries of the form $\langle \mathsf{x}, \mathsf{v} \rangle$, where x is the address of a shared variable and v is a version. The read log affords flexibility in its implementation and it can be implemented as a sequential store buffer (permitting duplicates) or as a set (prohibiting duplicates).

As in Section 3.3, we use T for the current thread, and t and t' for other threads.

Analysis at writes. Algorithm 3 shows the analysis that Valor performs at a write. It does nothing if x's last writer epoch matches the current thread T's current epoch (line 2), indicating that T has already written to x. Otherwise, the analysis checks for a write–write conflict (lines 3–4) by checking if c = clock(t), indicating that x was last written by an ongoing region in another thread (note that this situation implies $\mathsf{t} \neq \mathsf{T}$). Finally, the analysis updates $\mathcal{W}_x$ with an incremented version and the current thread's epoch (line 6).

Algorithm 3 WRITE [Valor]: thread T writes variable x

```
1: let ⟨v, c@t⟩ ← W_x
2: if c@t ≠ epoch(T) then                          ▷ Write in same region
3:     if c = clock(t) then
4:         Conflict!                               ▷ Write–write conflict detected
5:     end if
6:     W_x ← ⟨v+1, epoch(T)⟩                       ▷ Update write metadata
7: end if
```

Analysis at reads. Algorithm 4 shows Valor's read analysis. The analysis first checks for a conflict with a prior write in another thread's ongoing region (lines 2–3). Then, the executing thread adds an entry to its read log (line 5). The new entry consists of x's address and its current version v.

Algorithm 4 READ [Valor]: thread T reads variable x

```
1: let ⟨v, c@t⟩ ← W_x
2: if t ≠ T ∧ c = clock(t) then
3:     Conflict!                                   ▷ Write–read conflict detected
4: end if
5: T.readLog ← T.readLog ∪ {⟨x, v⟩}
```

Unlike FastRCD's analysis at reads (Algorithm 2), which updates FastRCD's read map $\mathcal{R}_x$, Valor's analysis at reads does not update any shared metadata. Valor thus avoids the synchronization and communication costs that FastRCD incurs updating read metadata.

Analysis at region end. Valor detects read–write conflicts lazily at region boundaries, as shown in Algorithm 5. For each entry $\langle \mathsf{x}, \mathsf{v} \rangle$ in the read log, the analysis compares v with x's current version v'. Differing versions are a necessary but insufficient condition for a conflict.

If x was last written by the thread ending the region, then a difference of *more* than one (i.e., $v' \geq v+2$) is necessary for a conflict (line 3).

Algorithm 5 REGION END [Valor]: thread T executes region boundary

```
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:         Conflict!                          ▷ Read–write conflict detected
5:     end if
6: end for
7: T.readLog ← ∅
```

We note that when Valor detects a write–write or write–read conflict, it is not necessarily the *first* conflict to occur: there may be an earlier read–write conflict waiting to be detected lazily. To report such read–write conflicts first, Valor triggers read validation before reporting a detected write–write or write–read conflict. As Section 3.4.3 explains, Valor can validate reads at any point to detect outstanding read–write conflicts, and does so before sensitive operations like system calls and I/O.

Atomicity of analysis operations. Similar to FastTrack and FastRCD, Valor's analysis at writes, reads, and region boundaries must execute atomically in order to avoid missing conflicts and corrupting analysis metadata. Unlike FastTrack and FastRCD, Valor can use a lock-free approach because the analysis accesses a single piece of shared state, $\mathcal{W}_x$. The write analysis updates $\mathcal{W}_x$ (line 6 in Algorithm 3) using an atomic operation (not shown). If the atomic operation fails because another thread updates $\mathcal{W}_x$ concurrently, the write analysis restarts from line 1. At reads and at region end, the analysis does not update shared state, so it does not need atomic operations.

3.4.3 Providing Valor's Guarantees

Like FastRCD, Valor soundly and precisely detects region conflicts. Section 3.7 proves that Valor is sound and precise.

Since Valor detects read–write conflicts lazily, it *cannot provide precise exceptions*. A read–write conflict will not be detected at the write, but rather at the end of the region that performed the read.

Deferred detection does *not* compromise Valor's semantic guarantees as long as the effects of conflicting regions do not become externally visible. A region that performs a read that conflicts with a later write can behave in a way that would be *impossible in any unserializable execution*. We refer to such regions as "zombie" regions, borrowing terminology from software transactional memory (STM) systems that experience similar issues by detecting conflicts lazily [88]. To prevent external visibility, Valor must validate a region's reads before all sensitive operations, such as system calls and I/O. Similarly, a zombie region might never end (e.g., might get stuck in an infinite loop), even if such behavior is impossible under any region serializable execution. To account for this possibility, Valor must periodically validate reads in a long-running region. Other conflict and data race detectors have detected conflicts asynchronously [54, 121, 167], providing imprecise exceptions and similar guarantees.

In an implementation for a memory- and type-unsafe language such as C or C++, a zombie region could perform arbitrary behavior such as corrupting arbitrary memory. This issue is problematic for lazy STMs that target C/C++, since a transaction can corrupt memory arbitrarily, making it impossible to preserve serializability [51]. The situation is not so dire for Valor, which detects region conflicts in order to throw conflict exceptions, rather than to preserve region serializability. As long as a zombie region does not actually corrupt

Valor's analysis state, read validation will be able to detect the conflict when it eventually executes—either when the region ends, at a system call, or periodically (in case of an infinite loop).

Our implementation targets a safe language (Java), so a zombie region's possible effects are safely limited.

3.5 Extending the Region Conflict Detectors

This section describes extensions that apply to both our proposed software-only region conflict detectors, FastRCD and Valor.

3.5.1 Demarcating Regions

The descriptions of FastRCD and Valor so far assumed that regions are demarcated by *all* synchronization operations, i.e., SFRs (Section 2.1). This is in tune with our goal of providing the SFRSx memory model. However, making regions larger helps detect more data races and can potentially help amortize fixed per-region costs. We argue that regions should be as large as possible, as long as any region conflict is guaranteed to be a data race. We observe that it is also correct to bound regions only at synchronization *release* operations (e.g., lock release, monitor wait, and thread fork) because region conflicts are still guaranteed to be true data races. We call these regions *release-free regions* (RFRs). Using RFRs as regions still allow FastRCD and Valor to soundly provide the SFRSx memory model.

Figure 2.3 showed an example of SFRs. Figure 3.3 illustrates the difference between SFRs and RFRs. We note that the boundaries of SFRs and RFRs are determined dynamically (at run time) by the synchronization operations that execute, as opposed to being determined statically at compile time. An RFR is *at least as large* as an SFR, so an RFR conflict detector

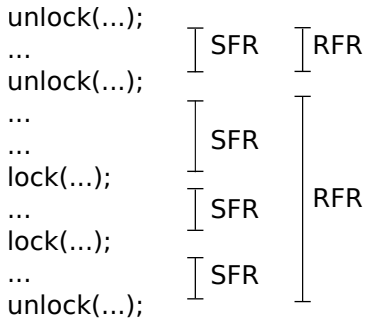


Figure 3.3: Synchronization- and release-free regions.

detects at least as many conflicts as an SFR conflict detector (as illustrated in Figures 3.4(a) and 3.4(b)). Larger regions potentially reduce fixed per-region costs, particularly the cost of updating writer metadata on the first write in each region.

There are useful analogies between RFR conflict detection and prior work. Happens-before data race detectors increment their epochs at release operations only [68, 146], and some prior work extends redundant instrumentation analysis past acquire, but not release, operations [71].

3.5.2 Correctness of RFR Conflict Detection

In this section, we prove that RFR conflicts correspond to true data races that can potentially violate region serializability. Recall that Valor and FastRCD define a region conflict as an access executed by one thread that conflicts with an already-executed access by another thread in an ongoing region. This definition is in contrast to an *overlap-based* region conflict definition that reports a conflict whenever two regions that contain conflicting accesses overlap at all. Both of these definitions support conflict detection between SFRs with no false positives. However, only the definition that we use for Valor and FastRCD supports conflict detection between RFRs without false data races; an overlap-based

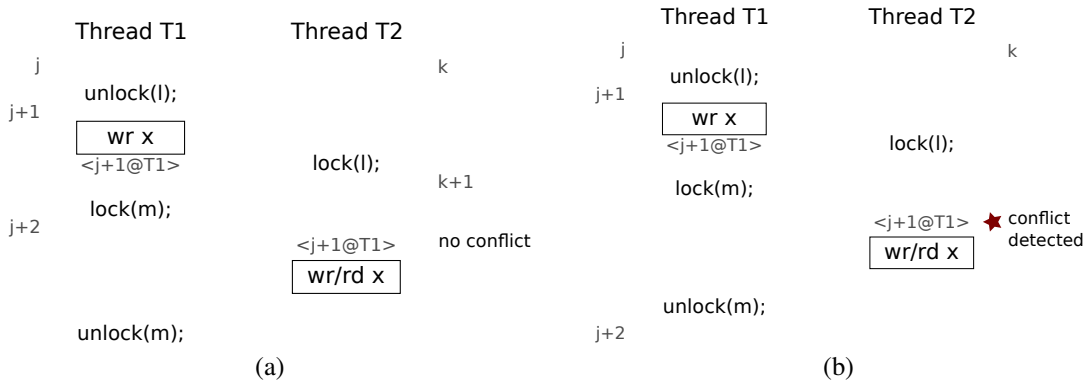


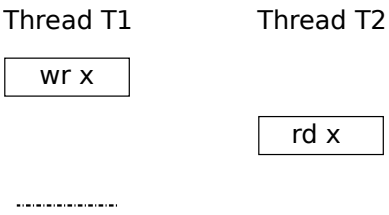
Figure 3.4: Detecting conflicts among RFRs allow more conflicts to be detected than SFRs, which are also true data races. (a) SFR j+1 in T1 has finished by the time T2 accesses x, and so no conflict is detected. (b) The same interleaving but with RFRs as regions is able to detect the conflict.

definition of RFR conflicts would yield false races. In the following, we prove the absence of false data races for our RFR conflict detection scheme.

RFR conflicts are data races. We prove the following theorem which applies to all of our proposed region conflict detectors:

Theorem 1. *Every release-free region (RFR) conflict is a true data race.*

Proof. We prove this claim by contradiction. Suppose that an RFR conflict exists in a DRF execution. Recall that, by definition, an RFR conflict exists when an access conflicts with another access executed by an ongoing RFR. Without loss of generality, we assume that a read by thread T2 conflicts with a write in an ongoing RFR in T1:



where the dashed line represents the end of the RFR that contains wr x.

Because we have assumed that the execution is DRF, T1's write must *happen before* T2's read:

$$\text{wr x} \prec_{HB} \text{rd x}$$

where $\prec_{HB}$ is the *happens-before* relation, a partial order that is the union of *program* order (i.e., intra-thread order) $\prec_{PO}$ and *synchronization* order $\prec_{SO}$ [104, 118].

Since wr x and rd x execute on different threads, they must be ordered in part by $\prec_{SO}$. Since $\prec_{SO}$ orders only synchronization operations, not ordinary reads and writes, wr x and rd x must also be ordered in part by $\prec_{PO}$. Furthermore, $\prec_{SO}$ can *only* order a release operation before an acquire operation (i.e., rel $\prec_{SO}$ acq). Thus, there must exist a release operation rel and an acquire operation acq such that

$$\text{wr x} \prec_{PO} \text{rel} \prec_{SO} \text{acq} \prec_{HB} \text{rd x}$$

Note that acq and rd x may be executed by different threads and/or be ordered by additional operations, so we cannot say anything more specific than acq $\prec_{HB}$ rd x.

The above ordering implies that rel is executed by T1 and that rel $\prec_{HB}$ rd x. Thus rd x does *not* overlap with the RFR that contains wr x, contradicting the initial assumption of an RFR conflict. □

3.5.3 Reporting Conflicting Sites

When a program executing under the "region conflict exception" memory model generates an exception, developers may want to know more about the conflict. We extend FastRCD and Valor to (optionally) report the source-level *sites*, which consist of the method and bytecode index (or line number), of both accesses involved in a conflict.

Data race detectors such as FastTrack report sites involved in data races by recording the access site alongside every thread–clock entry. Whenever FastTrack detects a conflict, it reports the corresponding recorded site as the first access and reports the current thread's site as the second access. Similarly, FastRCD can record the site for every thread–clock entry, and reports the corresponding site for a region conflict.

By recording sites for the last writer, Valor can report the sites for write–write and write–read conflicts. To report sites for read–write conflicts, Valor stores the read site with each entry in the read log. When it detects a conflict, Valor reports the conflicting read log entry's site and the last writer's site.

3.6 Alternate Metadata and Analysis for Valor

As presented in the last two sections, Valor maintains an epoch `c@t` (as well as a version `v`) for each variable `x`. An epoch enables a thread to query whether an ongoing region has written `x`. An alternate way to support that query is to track *ownership*: for each variable `x`, maintain the thread `t`, if any, that has an ongoing region that has written `x`. This section proposes an alternate design for Valor that tracks ownership instead of using epochs. For clarity, the rest of this chapter refers to the design of Valor described in Section 3.4 as *Valor-E* (Epoch) and the alternate design introduced here as *Valor-O* (Ownership).

We implement and evaluate *Valor-O* for implementation-specific reasons: (1) our implementation targets IA-32 (Section 3.8); (2) metadata accesses must be atomic (Section 3.4.2); and (3) Valor-O enables storing metadata (ownership and version) in 32 bits.

We do not expect either design to perform better in general. Valor-O consumes less space and uses slightly simpler conflict checks, but it incurs extra costs to maintain ownership: in

order to clear each written variable's ownership at region end, each thread must maintain a "write set" of variables written by its current region.

Metadata representation. Valor-O maintains a last writer tuple $\langle \mathsf{v}, \mathsf{t} \rangle$ for each shared variable x. The version v is the same as Valor-E's version. The thread t is the "owner" thread, if any, that is currently executing a region that has written x; otherwise t is ϕ.

Analysis at writes. Algorithm 6 shows Valor-O's analysis at program writes. If T is already x's owner, it can skip the rest of the analysis since the current region has already written x (line 2). Otherwise, if x is owned by a concurrent thread, it indicates a region conflict (lines 3–4). T updates x's write metadata to indicate ownership by T and to increment the version number (line 6).

Algorithm 6 WRITE [Valor-O]: thread T writes variable x

```
1: let ⟨v,t⟩ ← W_x
2: if t ≠ T then                                  ▷ Write in same region
3:     if t ≠ φ then
4:         Conflict!                              ▷ Write–write conflict detected
5:     end if
6:     W_x ← ⟨v+1,T⟩                              ▷ Update write metadata
7:     T.writeSet ← T.writeSet ∪ {x}
8: end if
```

A thread relinquishes ownership of a variable only at the next region boundary. To keep track of all variables owned by a thread's region, each thread T maintains a *write set*, denoted by $\mathsf{T}.\mathsf{writeSet}$ (line 7), which contains all shared variables written by T's current region.

Analysis at reads. Algorithm 7 shows Valor-O's analysis at program reads, which checks for write–read conflicts by checking x's write ownership (lines 2–3), but otherwise is the same as Valor-E's analysis (Algorithm 4).

Algorithm 7 READ [Valor-O]: thread T reads variable x

```
1: let ⟨v, t⟩ ← W_x
2: if t ≠ φ ∧ t ≠ T then
3:     Conflict!                                  ▷ Write–read conflict detected
4: end if
5: T.readLog ← T.readLog ∪ {⟨x, v⟩}
```

Analysis at region end. Algorithm 8 shows Valor-O's analysis for validating reads at the end of a region. To check for read–write conflicts, the analysis resembles Valor-E's analysis except that it checks each variable's owner thread, if any, rather than its epoch (line 3).

Algorithm 8 REGION END [Valor-O]: thread T executes region boundary

```
1: for all ⟨x, v⟩ ∈ T.readLog do
2:     let ⟨v', t⟩ ← W_x
3:     if (v' ≠ v ∧ t ≠ T) ∨ v' ≥ v + 2 then
4:         Conflict!                              ▷ Read–write conflict detected
5:     end if
6: end for
7: T.readLog ← ∅
8: for all x ∈ T.writeSet do
9:     let ⟨v, t⟩ ← W_x                           ▷ Can assert t = T
10:    W_x ← ⟨v, φ⟩                               ▷ Remove ownership by T
11: end for
12: T.writeSet ← ∅
```

Finally, the analysis at region end processes the write set by setting the ownership of each owned variable to ϕ (lines 8–11) and then clearing the write set (line 12).

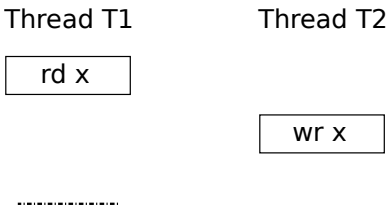
3.7 Valor Is Sound and Precise

This section proves that Valor detects region conflicts soundly and precisely.[9] That is, it reports a conflict if and only if an execution has a region conflict, which is defined as an access that conflicts with an access executed in an *ongoing* region (Section 3.2). Here we assume that the relatively straightforward FastRCD algorithm detects region conflicts soundly and precisely.

Theorem 2. *Valor is sound: if an execution has a region conflict, Valor reports a conflict.*

Proof. We prove the claim by contradiction. Suppose an execution has a region conflict and Valor reports no conflict.

Valor detects write–write and write–read conflicts identically to FastRCD, which we assume is sound, so Valor detects all write–write and write–read conflicts. Thus, the undetected conflict must be a read–write conflict. Without loss of generality, suppose thread T2 writes a variable x that conflicts with a region executed by thread T1. By the definition of region conflict, T2's write happens between T1's read to x and T1's region end:



where the dashed line indicates the earliest region boundary after T1's read, and the write is T2's first write to x after T1's read. Henceforth, "T1's region" and "T2's region" refer to the regions that contain the conflicting read and write, respectively.

At T1's read to x, let v be x's version (from $\mathcal{W}_x$). T1 logs $\langle x, v \rangle$ at the read (Algorithms 4 and 7). When T1's region ends, it performs read validation, which checks the following

[9]The theorems and proofs apply to both Valor-E and Valor-O.

condition for the read log entry $\langle x, v \rangle$ (Algorithms 5 and 8):

$$(v' \neq v \wedge t \neq T1) \vee v' \geq v+2$$

where v' and t are x's version and last-writer thread (from $\mathcal{W}_x$), respectively, at the time of validation.

Since our initial assumption was that Valor does not report a conflict, the condition must be false, i.e.,

$$v' = v \vee (t = T1 \wedge v' < v+2)$$

We consider each of the disjunction's predicates in turn.

Case 1: $v' = v$

Since Valor increments versions monotonically, $v' = v$ only if $T2$'s write does not increment x's version, which happens only if $T2$'s region has already written x (Algorithms 3 and 6). We assumed that $T2$'s write to x is the first write since $T1$'s read, so $T2$'s region must have written x *before* $T1$'s read. By definition of region conflict, a write–read region conflict exists, which Valor detects, contradicting the initial assumption.

Case 2: $t = T1 \wedge v' < v+2$

Since $t = T1$, $T1$ must be the last writer to x before read validation (Algorithms 3 and 6). The earliest such write must increment x's version unless $T1$ wrote x prior to $T2$'s write—but that would be a write–write conflict, contradicting the initial assumption. Similar to Case 1, $T2$'s write must increment x's version unless its region wrote x prior to $T1$'s read—but that would be a write–read conflict, contradicting the initial assumption. Thus, Valor must have incremented x's version at least twice, so $v' \geq v+2$, contradicting this case's premise.

Both cases lead to contradictions, so the assumption that Valor misses a conflict is false. □

Theorem 3. *Valor is precise: it reports a conflict only for an execution that has a region conflict.*

Proof. We prove the claim by contradiction. Suppose Valor reports a conflict for an execution that has no region conflict.

Valor detects and reports write–write and write–read conflicts identically to FastRCD, which we assume is precise, so the conflict must be a read–write conflict. Valor detects read–write conflicts only during read validation (Algorithms 5 and 8). Without loss of generality, suppose that thread T reports a conflict during read validation when validating a read log entry $\langle \mathsf{x}, \mathsf{v} \rangle$:

where the dashed line represents the earliest region boundary following the read.

Since read validation reports a conflict, the following condition must be satisfied:

$$(\mathsf{v}' \neq \mathsf{v} \wedge \mathsf{t} \neq \mathsf{T}) \vee \mathsf{v}' \geq \mathsf{v} + 2$$

where v' and t are x's version and last-writer thread (from $\mathcal{W}_x$), respectively, at the time of validation.

At least one of the two predicates of the disjunction must be satisfied:

Case 1: $\mathsf{v}' \neq \mathsf{v} \wedge \mathsf{t} \neq \mathsf{T}$

Because $\mathsf{v}' \neq \mathsf{v}$ (and only Valor's write analysis updates $\mathcal{W}_x$), there must have been a write by t to x between T's read and the region end that updated $\mathcal{W}_x$ to $\langle \mathsf{v}', \mathsf{c@t} \rangle$ (Valor-E's representation; Algorithm 3) or $\langle \mathsf{v}', \mathsf{t} \rangle$ (Valor-O's representation; Algorithm 6).

Based on our initial assumption of region conflict freedom, this write must have been executed by T. Thus, $\mathsf{t} = \mathsf{T}$, contradicting this case's premise.

Case 2: $\mathsf{v}' \geq \mathsf{v} + 2$

In order to increment x's version at least twice between T's read and region end, at least two writes in distinct regions must have written x (Algorithms 3 and 6). Only one of these writes can be in T's read's region, so the other write must be by a different thread, which by definition is a read–write region conflict, which contradicts the initial assumption.

Both cases lead to contradictions, so the assumption that Valor reports a false region conflict is false. □

3.8 Implementation

We have implemented FastTrack, FastRCD, and Valor in Jikes RVM 3.1.3 [9], a Java virtual machine (JVM) that performs competitively with commercial JVMs [20]. In the following, we first describe our implementation infrastructure, Jikes RVM. We will then present implementation details of FastTrack, FastRCD, and Valor in Jikes. Our implementations share features as much as possible: they instrument the same accesses, and FastRCD and Valor demarcate regions in the same way. As Section 3.6 mentioned, we implement the *Valor-O* design of Valor. We have made our implementations publicly available on the Jikes RVM Research Archive.[10]

[10] `http://www.jikesrvm.org/Resources/ResearchArchive/`

3.8.1 Jikes RVM: Our Implementation Infrastructure

Jikes RVM is a high-performance research JVM that is written in Java, and provides performance competitive with commercial JVMs [20].

Jikes RVM uses just-in-time compilation to generate machine code for each method at run-time. Jikes RVM has two dynamic compilers: *baseline* and *optimizing*. The baseline compiler is used to generate machine code when an application executes a method for the first time. The baseline compiler transforms bytecode to machine code directly, and does not perform any code optimizations. If a method is executed several times, it is considered "hot" and the VM recompiles the method using the optimizing compiler. The optimizing compiler transforms the bytecode to an internal representation (IR), and performs many optimizations such as inlining, constant propagation, and register allocation. The optimizing compiler supports different optimization levels. Our implementations modify both the dynamic compilers to add instrumentation to the application and library code.

Jikes RVM also supports automatic memory management using the *Memory Management Toolkit* [21][11] (MMTk), and implements several garbage collection (GC) policies.

More details about the architecture of Jikes RVM and instructions to setup and execute applications with Jikes RVM are available online.[12]

Evaluating competitiveness of Jikes RVM. To evaluate the competitiveness of Jikes RVM and to lend credibility to the performance results reported in this dissertation, this section compares the run-time performance of two JVMs, Jikes RVM 3.1.3 and OpenJDK 1.7. OpenJDK is a free and open source implementation of the Java Platform, Standard Edition (Java SE), and is the official reference implementation of Java SE starting from

[11] `http://jikesrvm.org/MMTk`

[12] `http://jikesrvm.org/Care+and+Feeding`

version 7. Figure 3.5 shows the relative performance of OpenJDK and Jikes RVM on two platforms: (a) a 64-core AMD system and (b) a 32-core Intel system. The two configurations, *OpenJDK* and *Jikes RVM*, represent unmodified OpenJDK and Jikes RVM respectively. The results are normalized to the first configuration.

As Figure 3.5 shows, overall Jikes RVM performs competitively with OpenJDK with one significant exception: pjbb2005, which performs 16.7X slower on both platforms, for reasons that are unknown to us. The two geomean bars show the geometric mean *including* and *excluding* pjbb2005 respectively. On average, Jikes RVM is 47% and 16% slower than OpenJDK on the AMD and Intel platforms, respectively. Excluding pjbb2005 from the geomean, Jikes RVM is 20% slower (AMD) and 7% faster (Intel) than OpenJDK.

By limiting execution to 32 cores (using the Linux taskset command) on the 64-core AMD machine (results not shown), we have concluded that most of the overhead difference between the two platforms is due to differences *other than* the core count, such as architectural differences.

These experiments suggest that although Jikes RVM was originally designed for research, it usually performs competitively with modern commercial JVMs, at least for our evaluated programs and platform.

3.8.2 Features Common to All Implementations

The implementations target IA-32 and extend Jikes RVM's *baseline* and *optimizing* dynamic compilers, to instrument synchronization operations and memory accesses. The implementations instrument all code in the *application context*, including application code and library code (e.g., java.*) called from application code.[13]

[13]Jikes RVM is itself written in Java, so both its code and the application code call the Java libraries. We have modified Jikes RVM to compile and invoke separate versions of the libraries for application and JVM contexts.

Instrumenting program operations. Each implementation instruments synchronization operations to track happens-before (FastTrack) or to demarcate regions (FastRCD and Valor). Acquire operations are lock acquire, monitor resume, thread start and join, and volatile read. Release operations are lock release, monitor wait, thread fork and terminate, and volatile write. By default, FastRCD and Valor detect conflicts between release-free regions (RFRs; Section 3.5.1) and add no instrumentation at acquires.

The compilers instrument each load and store to a scalar object field, array element, or static field, except in a few cases: (1) final fields, (2) volatile accesses (which we treat as synchronization operations), (3) accesses to a few immutable library types (e.g., String and Integer), and (4) redundant instrumentation points, as described next.

Eliminating redundant instrumentation. We have implemented an intraprocedural data-flow analysis to identify *redundant* instrumentation points. Instrumentation on an access to x is redundant if it is definitely preceded by an access to x in the same region (cf. [35, 71]). Specifically, instrumentation at a write is redundant if preceded by a write, and instrumentation at a read is redundant if preceded by a read or write. The implementations eliminate redundant instrumentation by default, which we find reduces *the run-time overheads added* by FastTrack, FastRCD, and Valor by 3%, 4%, and 5%, respectively (results not shown).

Tracking last accesses and sites. The implementations add last writer and/or reader information to each scalar object field, array element, and static field. The implementations lay out a field's metadata alongside the fields; they store an array element's metadata in a metadata array reachable from the array's header.

The implementations optionally include site tracking information with the added metadata. We evaluate data race coverage with site tracking enabled, and performance with site tracking disabled.

3.8.3 FastTrack and FastRCD

The FastRCD implementation shares many features with our FastTrack implementation, which is faithful to prior work's implementation [68]. Both implementations increment a thread's logical clock at each synchronization release operation, and they track last accesses similarly. Both maintain each shared variable's last writer and last reader(s) using FastTrack's epoch optimizations. In FastTrack, if the prior read is an epoch that *happens before* the current read, the algorithm continues using an epoch, and if not, it upgrades to a read map. FastRCD uses a read epoch if the last reader region has ended, and if not, it upgrades to a read map. Each read map is an efficient, specialized hash table that maps threads to clocks. We modify garbage collection (GC) to check each variable's read metadata and, if it references a read map, to trace the read map.

We represent FastTrack's epochs with two (32-bit) words. We use 9 bits for thread identifiers, and 1 bit to differentiate a read epoch from a read map. Encoding the per-thread clock with 22 bits to fit the epoch in one word would cause the clock to overflow, requiring a separate word for the clock.

FastRCD represents epochs using a single 32-bit word. FastRCD avoids overflow by leveraging the fact that it is always correct to reset all clocks to either 0, which represents a completed region, or 1, which represents an ongoing region. To accommodate this strategy, we modify GC in two ways. (1) Each full-heap GC sets every variable's clock to 1 if it

was accessed in an ongoing region and to 0 otherwise. (2) Each full-heap GC resets each thread's clock to 1. Note that FastTrack cannot use this optimization.

Despite FastRCD resetting clocks at every full-heap GC, a thread's clock may still exceed 22 bits. FastRCD could handle overflow by immediately triggering a full-heap collection, but we have not implemented that extension.

Atomicity of instrumentation. To improve performance, our implementations of FastTrack and FastRCD eschew synchronization on analysis operations that do not modify the last writer or reader metadata. When metadata must be modified, the instrumentation ensures atomicity of analysis operations by locking one of the variable's metadata words, by atomically setting it to a special value.

Tracking happens-before. In addition to instrumenting acquire and release synchronization operations as described in Section 3.8.2, FastTrack tracks the happens-before edge from each static field initialization in a class initializer to corresponding uses of that static field [111]. The FastTrack implementation instruments static (including final) field loads as an acquire of the same lock used for class initialization, in order to track those happens-before edges.

3.8.4 Valor

We implement the Valor-O design of Valor described in Section 3.6.

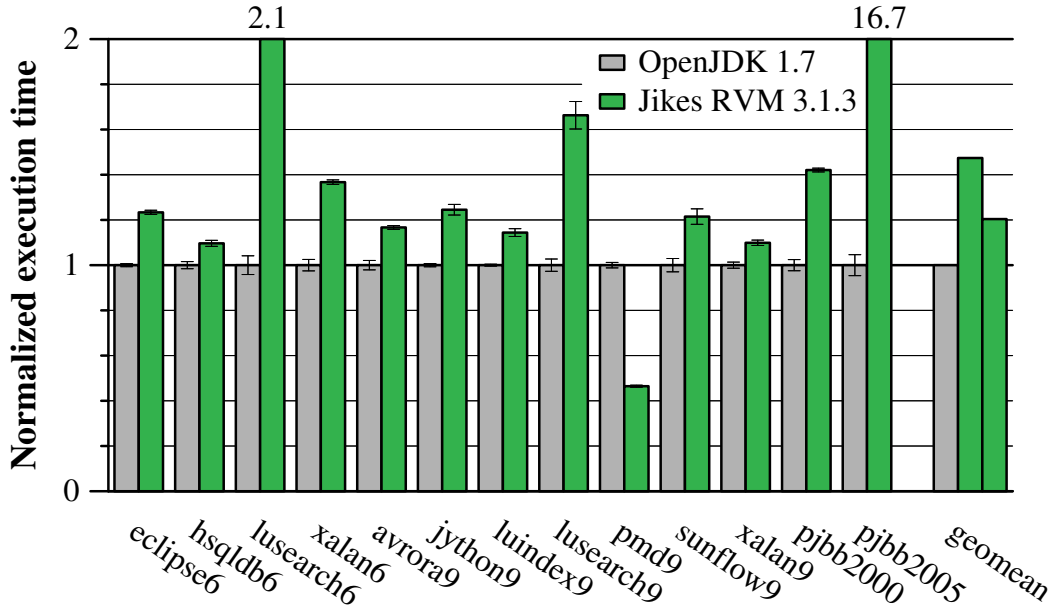
Tracking the last writer. Valor tracks the last writer in a single 32-bit metadata per variable: 23 bits for the version and 9 bits for the thread. Versions are unlikely to overflow because variables' versions are independent, unlike overflow-prone clocks, which are

updated at every region boundary. We find that versions overflow in only two of our evaluated programs. A version overflow could lead to a missed conflict (i.e., a false negative) if the overflowed version happened to match some logged version. To mitigate version overflow, Valor could reset versions at full-heap GCs, as FastRCD resets its clocks (Section 3.8.3).

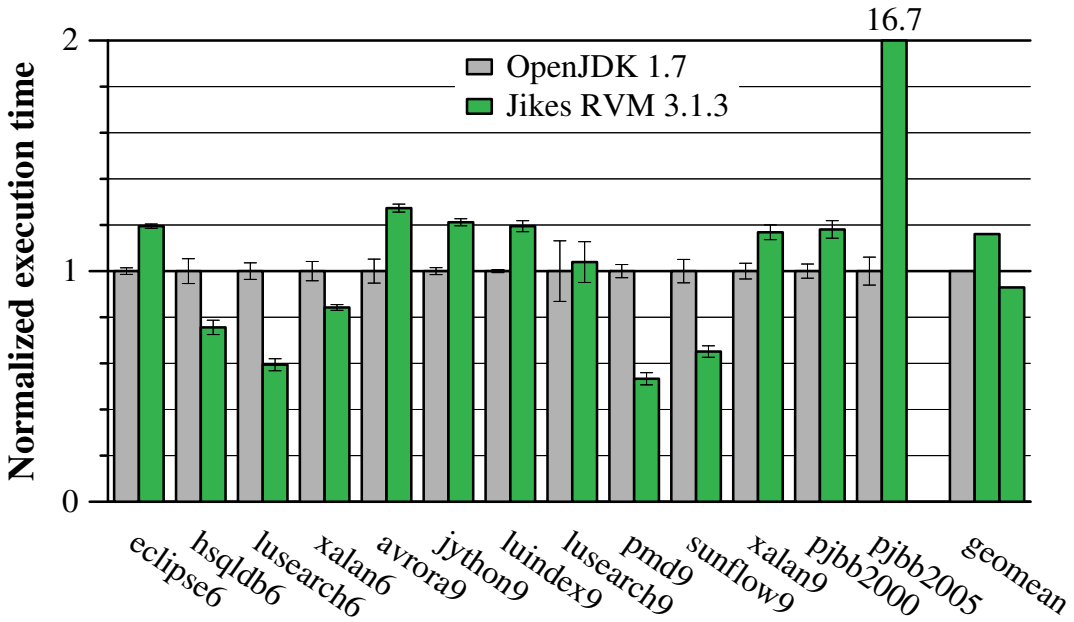
Access logging. We implement each per-thread read log as a sequential store buffer (SSB), so read logs may contain duplicate entries. Each per-thread write set is also an SSB, which is naturally duplicate free because only a region's first write to a variable updates the write set. To allow GC to trace read log and write set entries, Valor records each log entry's variable x as a base object address plus a metadata offset.

Handling large regions. A region's read log can become arbitrarily long because an executed region's length is not bounded. Our Valor implementation limits a read log's length to 2^{16} entries. When the log becomes full, Valor does read validation and resets the log.

The write set can also overflow, which is uncommon since it is duplicate free. When the write set becomes full ($>2^{16}$ elements), Valor conceptually splits the region by validating and resetting the read log (necessary to avoid false positives) and relinquishing ownership of variables in the write set.



(a) Performance on the 64-core AMD system.



(b) Performance on the 32-core Intel system.

Figure 3.5: Relative performance of OpenJDK and Jikes RVM on two platforms. In each graph, the two *geomean* bars for Jikes RVM are the geomean including and excluding pjbb2005.

3.9 Evaluation

This section evaluates and compares the performance and other characteristics of our implementations of FastTrack, FastRCD, and Valor.

3.9.1 Methodology

Benchmarks. We evaluate our implementations of Valor, FastRCD, and FastTrack using large, realistic, benchmarked applications: the *large* workload size of the multithreaded DaCapo benchmarks [22] that Jikes RVM 3.1.3 can execute successfully: eclipse6, hsqldb6, lusearch6, xalan6, avrora9, jython9, luindex9, lusearch9,[14] pmd9, sunflow9, and xalan9 (suffixes '6' and '9' distinguish benchmarks from versions 2006-10-MR2 and 9.12-bach, respectively). We also execute fixed-workload versions of SPECjbb2000 and SPECjbb-2005 programs, which we refer to as pjbb2000 and pjbb2005, respectively.[15] We omit single-threaded programs and programs that Jikes RVM 3.1.3 cannot execute.

Experimental setup. Each detector is built into a high-performance 32-bit JVM configuration called FastAdaptive that optimizes the JVM, adaptively optimizes the application code, and uses the default, high-performance, generational, stop-the-world garbage collector [23] (GenImmix). All experiments use a 64 MB nursery for generational GC, instead of the default 32 MB, because the larger nursery improves performance of all three detectors. The baseline (unmodified JVM) is negligibly improved on average by using a 64 MB nursery.

We limit the GC to 4 threads instead of the default 64 because of a known scalability bottleneck in Jikes RVM's memory management toolkit (MMTk) [55]. Using 4 GC threads

[14] We use a version of lusearch9 that fixes a serious memory leak [191].

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

improves performance for all configurations and the baseline. This change leads to reporting *higher* overheads for FastTrack, FastRCD, and Valor than with 64 GC threads, since less time is spent in GC, so the time added for conflict detection is a greater fraction of baseline execution time.

Platform. The experiments execute on an AMD Opteron 6272 system with eight 8-core 2.0-GHz processors (64 cores total), running RedHat Enterprise Linux 6.6, kernel 2.6.32.

We have also measured performance on an Intel Xeon platform with 32 cores, as summarized in Section 3.9.3.

3.9.2 Run-Time Overhead

Figure 3.6 shows the overhead added over unmodified Jikes RVM by the different implementations. Each bar is the average of 10 trials in order to minimize the effect of any machine noise. Each bar has a 95% confidence interval that is centered at the mean. *The main performance result in this work is that Valor incurs only 99% run-time overhead on average, far exceeding the performance of any prior conflict detection technique.* We discuss Valor's performance result in context by comparing it to FastTrack and FastRCD.

FastTrack. Our FastTrack implementation adds 342% overhead on average (i.e., 4.4X slowdown). Prior work reports an 8.5X average slowdown, but for a different implementation and evaluation [68]. In Section 3.9.8, we compare the two FastTrack implementations in our evaluation setting.

FastRCD. Figure 3.6 shows that FastRCD adds 267% overhead on average. FastRCD tracks accesses similarly to FastTrack, but has lower overhead than FastTrack because it

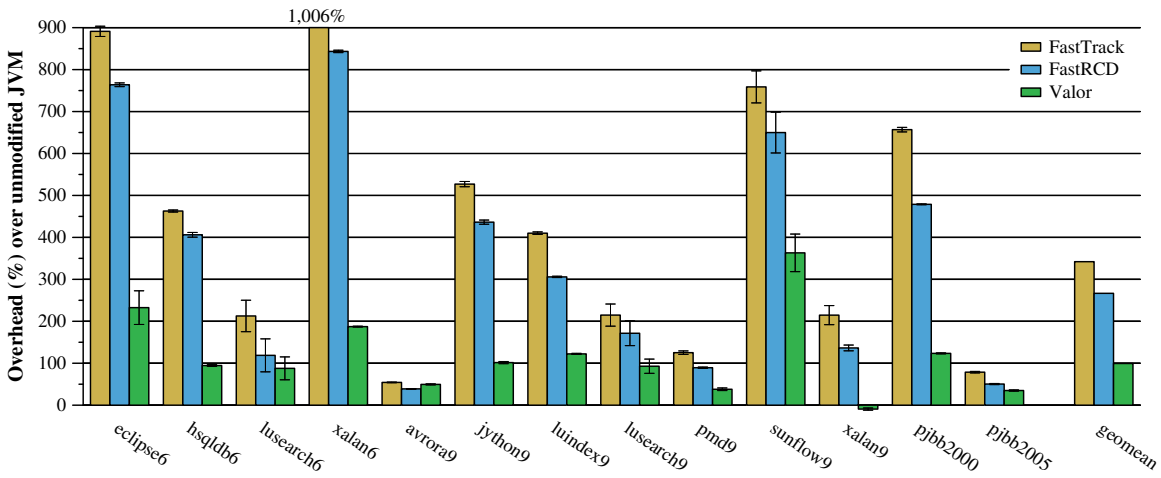


Figure 3.6: Run-time overhead added to unmodified Jikes RVM by our implementations of FastTrack, FastRCD, and Valor.

does not track happens-before. We measured that around 70% of FastRCD's cost comes from tracking last readers; the remainder comes from tracking last writers, demarcating regions, and bloating objects with per-variable metadata. Observing the high cost of last reader tracking motivates Valor's lazy read validation mechanism.

Valor. Valor adds only 99% overhead on average, which is substantially lower than the overheads of any prior software-only technique, including our FastTrack and FastRCD implementations. The most important reason for this improvement is that Valor completely does away with expensive updates and synchronization on last reader metadata. Valor consistently outperforms FastTrack and FastRCD for all programs except `avrora9`, for which FastTrack and FastRCD add particularly low overhead (for unknown reasons). Valor slightly outperforms the baseline for `xalan9`; we believe this unintuitive behavior is a side effect of reactive Linux thread scheduling decisions, as others have observed [14].

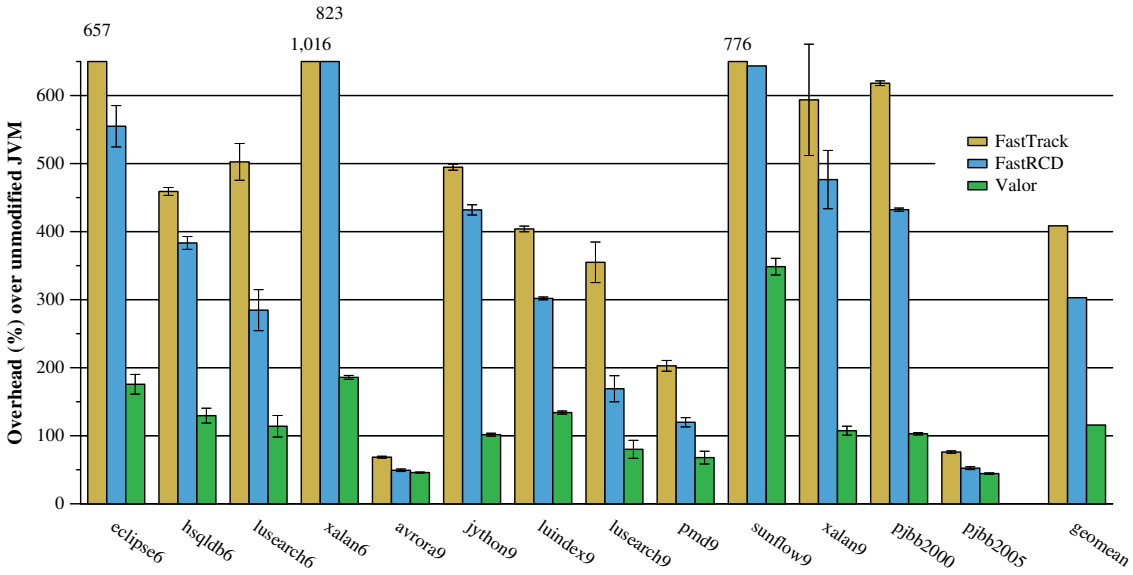


Figure 3.7: Run-time overhead added to unmodified Jikes RVM by our implementations of FastTrack, FastRCD, and Valor on an Intel Xeon E5-4620 system. Other than the platform, the methodology is the same as for Figure 3.6.

3.9.3 Architectural Sensitivity

This section evaluates the sensitivity of our experiments to the CPU architecture by repeating our performance experiments on an Intel Xeon E5-4620 system with four 8-core processors (32 cores total). Otherwise, the methodology is the same as in Section 3.9.2. Figure 3.7 shows the overhead added over unmodified Jikes RVM by our implementations. FastTrack adds an overhead of 408%, while FastRCD adds 303% overhead. Valor continues to substantially outperform the other techniques, adding an overhead of only 116%.

The *relative performance* of Valor compared to FastTrack and FastRCD is similar on both platforms. On the Xeon platform, Valor adds 3.5X and 2.6X less overhead than FastTrack and FastRCD, respectively, on average. On the (default) Opteron platform, Valor adds 3.4X and 2.7X less overhead on average.

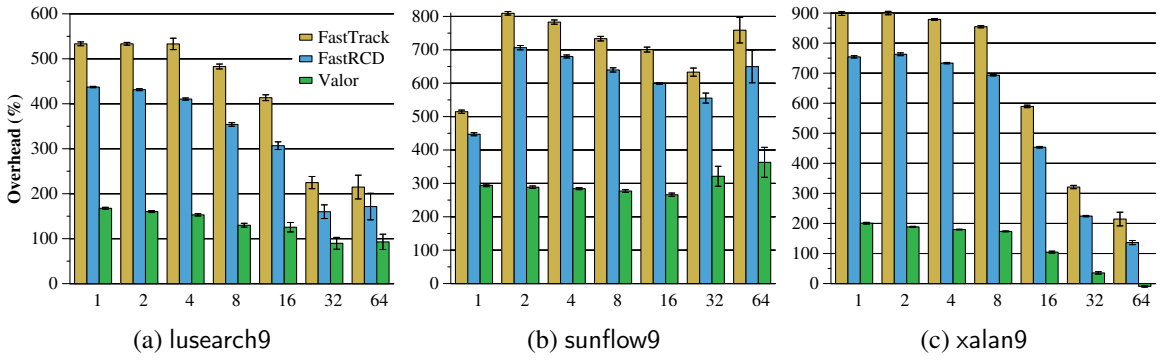


Figure 3.8: Run-time overheads of the configurations from Figure 3.6, for 1–64 application threads. The legend applies to all graphs.

3.9.4 Scalability

This section evaluates how the run-time overhead of Valor, compared with FastRCD and FastTrack, varies with additional application threads—an important property as systems increasingly provide more cores. We use the three evaluated programs that support spawning a configurable number of application threads: lusearch9, sunflow9, and xalan9 (Table 3.1). Both lusearch9 and sunflow9 naturally scale with unmodified Jikes RVM, while xalan9 starts anti-scaling after 32 threads (results not shown). Figure 3.8 shows the overhead for each program over the unmodified JVM for 1–64 application threads, using the configurations from Figure 3.6. Figure 3.8 shows that all three techniques' overheads scale with increasing numbers of threads. Valor in particular provides similar or decreasing overhead as the number of threads increases.

3.9.5 Space Overhead

This section evaluates the space overhead added by FastTrack, FastRCD, and Valor. We measure an execution's space usage as the maximum memory used after any full-heap

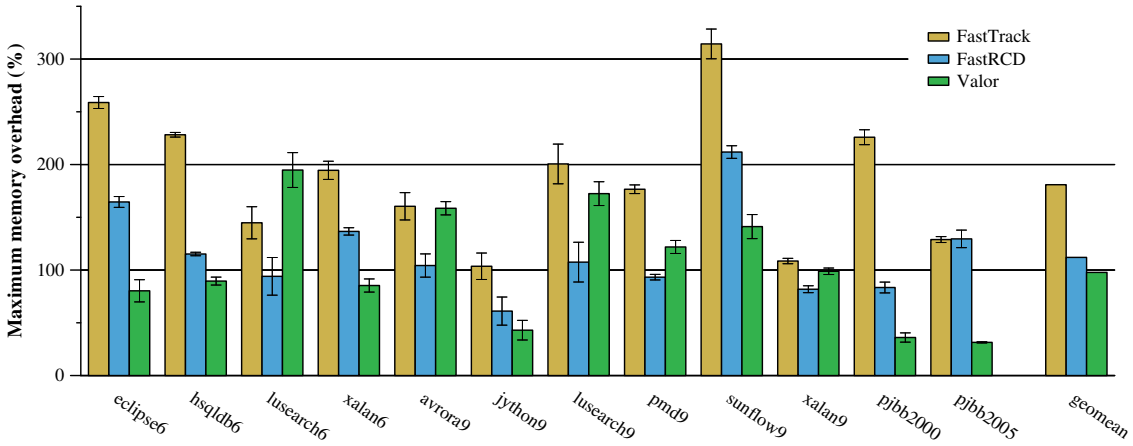


Figure 3.9: Space overheads of the configurations from Figure 3.6.

garbage collection (GC). Our experiments use Jikes RVM configured with the default, high-performance, generational GC and let the GC adjust the heap size automatically (Section 3.9.1).

Figure 3.9 shows the space overhead, relative to baseline (unmodified JVM) execution for the same configurations as in Figure 3.6. We omit luindex9 since the unmodified JVM triggers no full-heap GCs, although each of the three analyses does. FastTrack, FastRCD, and Valor add 180%, 112%, and 98%, respectively. Unsurprisingly, FastTrack uses more space than FastRCD since it maintains more metadata. Valor sometimes adds less space than FastRCD; other times it adds more. This result is due to the analyses' different approaches for maintaining read information: FastRCD uses per-variable shared metadata, whereas Valor logs reads in per-thread buffers. On average, Valor uses less memory than FastRCD and a little more than half as much memory as FastTrack.

3.9.6 Run-Time Characteristics

Table 3.1 characterizes the evaluated programs' behavior. Each value is the mean of 10 trials of a statistics-gathering version of one of the implementations. The first two columns report the total threads created and the maximum active threads at any time.

The next columns, labeled *Reads* and *Writes*, report instrumented, executed read and write operations (in millions). The *No metadata updates* columns show the percentage of instrumented accesses for which instrumentation need not update or synchronize on any metadata. For FastTrack, these are its "read/write same epoch" and "read shared same epoch" cases [68]. For FastRCD and Valor, these are the cases where the analysis does not update any per-variable metadata. Note that Valor has no *Reads* column because it does not update per-variable metadata on a program read.

For three programs, FastTrack and FastRCD differ significantly in how many reads require metadata updates (minor differences for other programs are not statistically significant). These differences occur because the analyses differ in when they upgrade from a read epoch to a read map (Section 3.8.3). For per-*write* metadata updates, the analyses report very similar percentages, so we report a single percentage (the percentage reported by FastTrack).

The last two columns report (1) how many release-free regions (RFRs), in thousands, each program executes and (2) the average number of memory accesses executed in each RFR. The RFR count is the same as the number of synchronization release operations executed and FastTrack's number of epoch increments. Most programs perform synchronization on average at least every 1,500 memory accesses. The outlier is sunflow9: its worker threads perform mostly independent work, with infrequent synchronization.

	Threads		Reads	Writes	No metadata updates (%)			Dyn. RFRs	Avg. accesses per RFR
					Reads		Writes		
	Total	Max live	($\times 10^6$)	($\times 10^6$)	FastTrack	FastRCD		($\times 10^3$)	
eclipse6	18	12	11,200	3,250	80.4	74.4	64.2	196,000	71
hsqldb6	402	102	575	79	41.5	41.6	13.8	7,600	86
lusearch6	65	65	2,300	798	83.4	83.5	79.4	9,880	311
xalan6	9	9	10,100	2,150	43.4	42.1	23.4	288,000	41
avrora9	27	27	4,790	2,430	88.6	88.7	91.9	6,340	1,133
jython9	3	3	4,660	1,370	59.1	48.9	38.3	199,000	28
luindex9	2	2	326	98	86.3	85.0	70.6	267	1,480
lusearch9*	64	64	2,360	692	84.3	84.5	77.3	6,050	494
pmd9	5	5	570	188	85.4	85.4	72.1	2,130	346
sunflow9*	128	64	19,000	2,050	95.4	95.4	47.9	10	2,140,000
xalan9*	64	64	9,317	2,100	52.0	51.2	28.2	108,000	106
pjbb2000	37	9	1,380	537	32.9	33.8	9.2	128,000	15
pjbb2005	9	9	6,140	2,660	54.9	37.6	9.7	283,000	30

Table 3.1: Run-time characteristics of the evaluated programs, executed by implementations of FastTrack, FastRCD, and Valor. Counts are rounded to three significant figures and the nearest whole number. Percentages are rounded to the nearest 0.1%. *Three programs by default spawn threads in proportion to the number of cores (64 in most of our experiments).

3.9.7 Data Race Detection Coverage

FastTrack detects every data race in an execution. In contrast, Valor and FastRCD focus on supporting conflict exceptions, so they detect only region conflicts, not all data races. That said, an interesting question is how many data races Valor and FastRCD detect compared with a fully sound data race detector like FastTrack. That is, how many data races manifest as region conflicts in typical executions?

Table 3.2 shows how many data races each analysis detects. A data race is defined as an unordered pair of static program locations (Section 2.1). If the same race is detected multiple times in an execution, we count it only once. The first number for each detector is the average number of races (rounded to the nearest whole number) reported across 10 trials. Run-to-run variation is typically small: 95% confidence intervals are consistently smaller than $\pm$10% of the reported mean, except for xalan9, which varies by $\pm$35% of the mean. The number in parentheses is the count of races reported at least once across all 10 trials.

As expected, FastTrack reports more data races than FastRCD and Valor. On average across the programs, one run of either FastRCD or Valor detects 58% of the true data races. Counting data races reported at least once across 10 trials, the percentage increases to 63% for FastRCD and 73% for Valor, respectively. Compared to FastTrack, FastRCD and Valor represent lower coverage, higher performance points in the performance–coverage tradeoff space. We note that FastRCD and Valor are *able* to detect any data race, because any data race can manifest as a region conflict [56].

We emphasize that although FastRCD and Valor miss some data races, the reported races involve accesses that are dynamically "close enough" together to jeopardize region serializability (Section 2.2.2). We (and others [56, 115, 121]) argue that region conflicts are therefore *more* harmful than other data races, and it is more important to fix them.

	FastTrack		FastRCD		Valor	
eclipse6	37	(46)	3	(7)	4	(21)
hsqldb6	10	(10)	10	(10)	9	(9)
lusearch6	0	(0)	0	(0)	0	(0)
xalan6	12	(16)	11	(15)	12	(16)
avrora9	7	(7)	7	(7)	7	(8)
jython9	0	(0)	0	(0)	0	(0)
luindex9	1	(1)	0	(0)	0	(0)
lusearch9	3	(4)	3	(5)	4	(5)
pmd9	96	(108)	43	(56)	50	(67)
sunflow9	10	(10)	2	(2)	2	(2)
xalan9	33	(39)	32	(40)	20	(39)
pjbb2000	7	(7)	0	(1)	1	(4)
pjbb2005	28	(28)	30	(30)	31	(31)

Table 3.2: Data races reported by FastTrack, FastRCD, and Valor. For each analysis, the first number is average distinct races reported across 10 trials. The second number (in parentheses) is distinct races reported at least once over all trials.

Although FastRCD and Valor both report RFR conflicts soundly and precisely, they may report different pairs of sites. For a read–write race, FastRCD reports the first read in a region to race, along with the racing write. If more than two memory accesses race, Valor reports the site of all reads that race, along with the racing write. As a result, Valor reports *more races than FastTrack* in a few cases because Valor reports multiple races between one region's write and another region that has multiple reads to the same variable x, whereas FastTrack reports only the read–write race involving the region's first read to x. We have manually verified that the reported races are in fact true data races.

Comparing SFR and RFR conflict detection. FastRCD and Valor bound regions at releases only, potentially detecting more races at lower cost as a result. We have evaluated the benefits of using RFRs in Valor by comparing with a version of Valor that uses SFRs. For every evaluated program, there is no statistically significant difference in races detected

between SFR- and RFR-based conflict detection (10 trials each; 95% confidence). RFR-based conflict detection does, however, outperform SFR-based conflict detection, adding 99% versus 104% overhead on average, respectively. This difference is due to RFRs being larger and thus incurring fewer metadata and write set updates (Section 3.5.1).

3.9.8 Comparing FastTrack Implementations

To fairly and directly compare FastTrack, FastRCD, and Valor, we have implemented all three approaches in Jikes RVM (Section 3.8). This section seeks to better understand the performance differences between our FastTrack implementation and Flanagan and Freund's publicly available FastTrack implementation [68].[16] Their FastTrack implementation is built on the *RoadRunner* dynamic bytecode instrumentation framework, which alone slows programs by 4–5X on average [68, 70]. We execute the RoadRunner FastTrack implementation on a different JVM, Open JDK 1.7, because Jikes RVM would not execute it correctly. RoadRunner does not fully support instrumenting the Java libraries (e.g., `java.*`), and it does not support the class loading pattern used by the DaCapo harness [76], so we are only able to execute a few programs successfully, and we exclude library instrumentation. (Recent work runs the DaCapo benchmarks successfully with RoadRunner by running the programs after extracting them from the harness [187].)

Figure 3.10 shows how the implementations compare for the programs that RoadRunner executes. For each program, the first two configurations execute with OpenJDK, and the last two execute with Jikes RVM. The results are normalized to the first configuration, which is unmodified OpenJDK. The second configuration, *RR + FT*, shows the slowdown that FastTrack (including RoadRunner) adds to OpenJDK. This slowdown is 9.3X, which is

[16] `https://github.com/stephenfreund/RoadRunner`

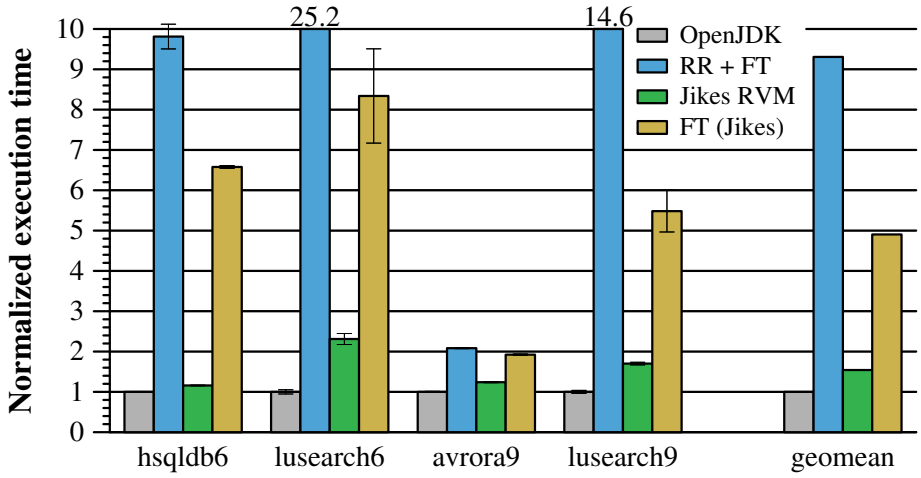


Figure 3.10: Performance comparison of FastTrack implementations. The last two configurations correspond to the baseline and FastTrack configurations in Figure 3.6.

close to the 8.5X slowdown reported by the FastTrack authors [68] in their experiments (with different programs on a different platform).

The last two configurations, *Jikes RVM* and *FT (Jikes)*, are the baseline and FastTrack configurations, respectively, from Figure 3.6. Note that this experiment keeps library instrumentation enabled for the last configuration, *FT (Jikes)*. Our FastTrack implementation in Jikes RVM adds significantly less overhead than the RoadRunner implementation, presumably because the Jikes RVM implementation is *inside* the JVM, so it can add efficient per-field and per-object metadata, modify the garbage collector, and control the compilation of instrumentation. In contrast, RoadRunner is a general framework that is implemented *on top of* the JVM using dynamic bytecode instrumentation.

For these four programs, unmodified Jikes RVM is about 54% slower than unmodified OpenJDK. (Section 3.8.1 compares the JVMs' performance across all programs.)

3.9.9 Summary

Overall, Valor substantially outperforms both FastTrack and FastRCD, adding, on average, just a third of FastRCD's overhead. Valor's overhead is potentially low enough for use in alpha, beta, and in-house testing environments and potentially even some production settings, enabling more widespread use of applying sound and precise region conflict detection to provide semantics to racy executions.

3.10 Contributions and Impact

In Section 2.2.1, we highlighted that current shared-memory programming languages such as C/C++ and Java provide no or weak semantics for racy executions. Our goal in this work is to develop efficient mechanisms that equip present and future languages with clear, intuitive semantics even for programs that permit data races. Recent work gives fail-stop semantics to data races, treating a data race as an exception [41, 115]. Our work is motivated by these efforts, and our techniques also give data races fail-stop semantics. This work introduces two new software-based region conflict detectors, FastRCD and Valor, to strengthen memory models and provide SFR serializability to all program executions. The key insight behind Valor is that detecting read–write conflicts lazily retains necessary semantic guarantees and has better performance than eager conflict detection.

Our proposed software-only technique, Valor, represents an advance in the state of the art for providing *always-on* support for strong guarantees for racy executions and for detecting data races. Valor has overheads low enough to provide practical semantic guarantees (i.e., SFR serializability) to a language specification. For example, future language runtimes can integrate an analysis like Valor to warn developers about potential violations of region serializability. This advance helps make it practical to use all-the-time conflict exceptions in

various settings, from in-house testing to alpha and beta testing to even some production systems. Using an analysis like Valor will help developers identify and eliminate *most or all* data races that lead to serializability violations during testing. For data races that escape testing and manifest in the wild, Valor will terminate program executions on production systems before allowing bad behavior to actually happen. In the longer term, this should translate to ease of development and better debugging of data races for programmers, and thus to safer and more reliable software.

Chapter 4: RCC: Practical Architecture Support for Region-Serializability-Based Consistency

4.1 Introduction

Chapter 3 presented a software-only solution called Valor to efficiently provide the SFRSx memory model to all program executions. A software-only solution such as Valor has the advantage of being immediately useful to several application environments. Although our proposed technique Valor is the fastest known region conflict detector and has overheads competitive enough to provide practical semantic guarantees to a language specification, it still slows programs by 2X which may not be suitable for use in all production environments. Our position in this work is that systems *should* provide strong, end-to-end memory models—but in order to gain traction, these mechanisms must incur costs and complexity on par with current weak memory model mechanisms.

Since region conflict detection can be sped up with hardware support, an alternate option to provide strong memory consistency models such as SFRSx is to explore optimal architectural support. This work describes a new, generally applicable conflict detection mechanism, realized in an efficient, complexity-effective architecture design. We illustrate the value of our new conflict detection mechanism by describing its application to supporting a memory consistency model called *SFRSx* [20, 115]. Even in an execution with data races,

SFRSx guarantees the serializability of *synchronization-free regions* (SFRs) of code, or halts with a *consistency exception* that indicates a conflict.

A feasible SFRSx implementation requires a precise, efficient, complexity-effective conflict detection mechanism—a requirement that has eluded prior work. Software-only approaches slow programs by 2X or more [20, 39, 52, 73, 89, 166]. *Hardware transactional memory* (HTM) systems limit region length [91, 193], require a software fallback [12, 38, 102, 127], or pay a high cost to support unbounded regions [10, 44, 85, 88, 148]. TM also usually tracks conflicts imprecisely, which is sufficient for speculative execution, but not for a precise SFRSx implementation. *Transactional Coherence and Consistency* (TCC) leverages TM to enforce coherence and consistency, but relies on expensive read set broadcasts, and is not efficiently scalable to unbounded regions [85] (Sections 4.6.5 and 6). Hardware support for SFRSx called *Conflict Exceptions* (CE) [115] places impractically high demands on the memory system to store and access analysis information (Section 4.6). Furthermore, CE is built on top of current systems' cache coherence mechanisms that serialize concurrent memory accesses to a single memory location, incurring the latency and complexity of a mechanism such as the M(O)ESI coherence protocol [141, 172]—which prior work showed is unnecessary for enforcing strong memory consistency [48, 85, 100]. These prior efforts thus are non-starters for high-performance parallel systems.

In this work, we explore an optimal architectural solution to provide end-to-end SFR serializability to all program executions efficiently. This work introduces *Region Consistency and Coherence* (RCC): a precise, efficient, complexity-effective architecture design that supports SFRSx by detecting conflicts between unbounded regions. RCC's contribution is a novel mechanism for detecting conflicts by ensuring write atomicity and checking read consistency. Our work makes a case for rethinking cache coherence and consistency

in providing SFRSx. RCC eliminates the need for legacy coherence protocol support, making better use of hardware resources, yielding a design that is both performant and complexity-effective. CE, which also provides SFRSx, is built on top of current systems' cache coherence mechanisms that serialize concurrent memory accesses to a single memory location, incurring the latency and complexity of a mechanism such as the M(O)ESI coherence protocol [141, 172]—which we show in this work is superfluous in a setting that ensures strong memory consistency.

4.2 Hardware Memory Models and Cache Coherence Protocols

While language memory models provide virtually no guarantees for executions with data races, nearly all *architectures* enforce comparatively strong guarantees. Still, pervasive hardware memory models used by x86 [164], SPARC [174], Power [157], and ARM [8] are weak, they lack clear race semantics and end-to-end guarantees permitting visible reordering of memory accesses [8, 157, 164, 174]. Furthermore, their guarantees are *not end-to-end*, i.e., they are with respect to the compiled program only. In the words of Adve and Boehm [2], "a strong hardware model is not very useful to programmers using languages and compilers that provide only a weak guarantee." Therefore, it is of little benefit in providing strong guarantees end-to-end in the presence of already-performed language-level reorderings in racy executions.

Cache coherence protocols. Most architectures use a *cache coherence protocol*, such as MESI [141], to guarantee cache coherence. These protocols typically enforce the *single writer / multiple readers* (SWMR) invariant [172], which provides the most recently written value for a location at all times. Adhering to the SWMR guarantee, a cache coherence protocol implementation furnishes reads with correct, up-to-date values.

A cache coherence protocol spends time, area, energy, on-chip bandwidth, and complexity to ensure cache lines are coherent. In a modern, directory-based, write-invalidate protocol (i.e., MESI or MOESI), the protocol maintains a *directory* of sharers for each line, which consumes area and increases total power. When a core accesses a cache line, the coherence protocol sends messages between cores and the directory to enforce the SWMR invariant. Before a core writes a line, it reads a list of sharers from the directory, sends an invalidation message to the sharers and collects acknowledgments of the invalidation from each sharer. Messaging increases power requirements, incurs latency, and increases on-chip network traffic, consuming valuable bandwidth. In addition to the run-time costs of a coherence protocol, even straightforward protocols like MESI have very high implementation complexity, which stems from the need for *transient protocol states* [172]. Some transient states exist only to provide coherence semantics to data races [48, 172], to which language memory models already ascribe undefined semantics [2, 31, 118].

For the properties they provide, cache coherence protocols like MESI offer good performance and scalability [123]. However, cache coherence alone is a relatively weak consistency property that does not match programming languages well. Our work's goal is to provide strong consistency that subsumes coherence and is a better match for what languages need. A key insight of our work is that by introducing support for strong region-based *consistency*, we can eliminate support for *coherence* provided eagerly at each memory access. We advocate for a departure from eager, access-granular coherence protocols, instead providing one mechanism that uniformly enforces region-based consistency and coherence properties for all accesses (including data races) and permits aggressive compiler and hardware reordering.

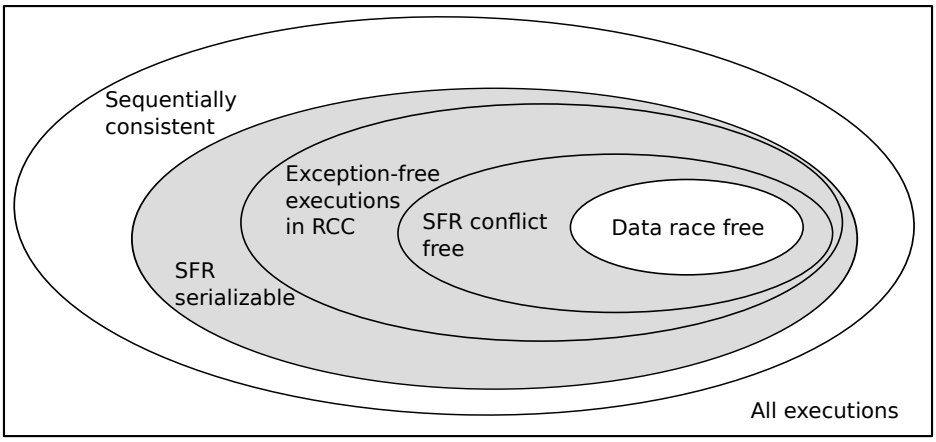


Figure 4.1: Relationship of possible execution behaviors. Behaviors that may or may not generate an exception under SFRSx are shaded gray. A memory model would restrict behaviors to one of these sets of executions.

Figure 4.1 shows the relationship between various types of execution behavior (regardless of the memory model). For example, if an execution is data race free, it is SFR serializable. If an execution is SFR serializable, it is SC. As prior work has motivated [20, 77, 115], SFRSx allows for some flexibility: an execution with a data race may or may not throw a consistency exception. In the figure, behaviors for which SFRSx may or may not generate an exception are shaded gray.

4.3 Design Overview of RCC

This chapter introduces a new architecture design called *Region Consistency and Coherence* (RCC) that provides the SFRSx memory model end-to-end, ensuring strong semantics for *all* program executions. RCC targets an existing memory model called SFRSx that either ensures serializability of SFRs or (only if there is a data race) reports a consistency exception [20, 115]. In contrast to prior efforts [115], RCC jettisons a traditional coherence mechanism, in favor of new hardware structures of similar complexity. In RCC, cores

execute largely independently, eschewing eagerly propagating memory access information throughout the system (e.g., to a directory or other cores), avoiding the latency of sending coherence messages and waiting for acknowledgments. This implies that the RCC protocol need not maintain read sharer information. Not having to maintain sharers also implies that RCC can do away with the necessity of requiring inclusivity for the last-level cache (LLC). Instead, RCC provides coherence and consistency *only at SFR boundaries (synchronization operations) and on cache line evictions*. RCC ensures SFRSx without incurring the full cost of detecting all region-to-region conflicts, by using a novel design that ensures serializability through mechanisms that detect some conflicts directly and infer others indirectly. By providing SFRSx, which is a stronger property than cache coherence alone, our architecture features are a better target for current and future language memory models than the pervasive, but mismatched, cache coherence property.

RCC's coherence mechanisms are inspired by prior work on distributed shared memory (DSM) systems that provide *release consistency* [3, 15, 25, 40, 64, 101] and prior work that simplifies cache coherence mechanisms [48, 58, 100, 154, 176, 177]. However, all of that work applies to DRF executions only, whereas RCC requires a substantially different design to ensure strong consistency for all executions (Section 6).

In RCC, if synchronization operations are frequent, enforcing coherence and consistency can be costly. We eliminate almost all of this cost with several effective optimizations. RCC's optimizations take a cue from prior approaches that use *self-invalidation* [48, 58, 100, 154] with the important and substantial difference that *these prior approaches provide no guarantees for executions with data races* (Section 6).

This section provides a high-level, view of the goals of RCC and an architecture-independent description of how RCC meets its goals. Sections 4.4 and 4.5 describe the RCC

architecture and optimizations. Section 4.6 discusses steps for simulating RCC and presents results. We conclude in Section 4.7.

4.3.1 RCC's Goals and Guarantees

Region Consistency and Coherence (RCC) is an architectural design that provides the SFRSx memory model end-to-end. In particular, RCC enforces the SFRSx memory model (Section 2.2.2.2), which ensures that every execution either (1) is equivalent to a serialization of SFRs; or (2) has a data race and generates a *consistency exception*. A consistency exception indicates that RCC has detected a conflict between SFRs (i.e., a true data race) that may jeopardize serializability. RCC requires no compiler or language-level changes: compilers already limit reordering across synchronization operations, so supporting SFRSx in hardware is sufficient to provide end-to-end guarantees. RCC treats synchronization operations (e.g., atomic instructions and memory fences) as SFR boundaries, permitting it to run legacy code. Alternatively, RCC could add an ISA extension (like the endR instruction from Conflict Exceptions [115]) to explicitly demarcate region boundaries in the instruction stream, at the expense of legacy support.

The rest of this chapter uses "region" and "SFR" interchangeably.

4.3.2 Overview and Insights

RCC provides architecture support for consistency (SFRSx) that avoids the need for cache coherence at the granularity of memory accesses. Instead RCC defers coherence until synchronization operations and private cache evictions.

In RCC, each core preserves SFR serializability by *executing each SFR in isolation and ensuring its consistency at SFR boundaries*. Each core tracks its reads and writes locally, in its private cache, during an SFR's execution. (For simplicity, this section assumes each core

has a single private cache.) When its SFR ends, the core ensures consistency by *committing* its writes atomically and *validating* that its reads are consistent with the values in the shared cache (i.e., the LLC). A detected conflict or a failure to validate a read indicates an SFR conflict and thus a possible violation of SFR serializability, so RCC generates a consistency exception.

With finite caches, a core cannot always cache an SFR's working set. When a region suffers a capacity miss, the core *delegates further consistency checking for that line to the LLC*. The LLC immediately checks for conflicts and validates that the line's reads are consistent with the values in the LLC. For the rest of the core's SFR, the LLC tracks the reads and writes for the evicted line, and it detects intervening conflicts from other cores.

By ensuring consistency (for non-evicted lines) at region boundaries by validating reads and committing writes atomically, RCC avoids the costs prior work has incurred to detect conflicts, e.g., by piggybacking on coherence messages to communicate read and write sets among cores [85, 115, 121, 167].

4.3.3 Design Details

State. Cores' private caches and the LLC track *access information* that represents, for each byte in a cache line, whether it has been read and/or written by a core's ongoing SFR. Byte-granular tracking is essential for *precise* conflict detection. The LLC maintains access information only for the lines evicted from a private cache to the LLC.

Each shared cache line maintains a *version* that is a monotonically increasing number, incremented on each write-back to the line in the LLC. A version represents the logical time of write-backs to the line in the LLC. When a shared line is fetched by a private cache, the

version is cached privately. A private cache does not update the version, but rather uses it to validate reads from the private line.

Actions at reads and writes. When a core reads or writes a byte of memory, it updates its private cache line's access information (read or write bit) for the accessed byte. If the byte was previously written and is being read, the core need not update the information. If a core evicts a line that has access information from its private cache, the private cache writes back the access information to the LLC, along with the line data (as usual) if the line is dirty. A dedicated *consistency controller* (Section 4.4) (CC) co-located with the LLC maintains per-core access information for evicted lines. The LLC uses the access information to detect conflicts with other cores when they validate reads and commit writes to the LLC, and when they evict lines with access information to the LLC. If a private cache fetches a line that was accessed by its current SFR but was then evicted, the private cache restores the same access information from the LLC.

Actions at SFR boundaries. When a core's SFR ends, it ensures serializability by validating its reads and committing its writes atomically, accomplished by performing the following three operations *in order*:

(1) Pre-commit: The core writes back dirty bytes to the LLC. The LLC checks for conflicts (which generate a consistency exception) using its access information for the written-back lines, and it updates access information for the committing core's lines. The LLC maintains the written-back lines' access information during the next step, read validation, to ensure the atomicity of validating reads and committing writes.

(2) Read validation: The core needs to validate that the values it read are consistent with the current values in the LLC. Sending data values would generate a lot of traffic, and comparing values alone could miss consistency violations by not validating with respect to a single memory snapshot. The core thus compares the *version* of each line read with the line in the LLC. To preserve soundness, the LLC checks for write–read conflicts during validation, even on a version match, if the shared line has write information set for a remote core.

Algorithm 9 shows how read validation works. For each cache line that needs to be validated, the private cache sends the line's version to the LLC for comparison. For simplicity, the algorithm depicts a synchronous reply for every validation request; in fact, the LLC's reply can be asynchronous, and it need not reply if the versions match (Section 4.4).

A version mismatch indicates a write to the same line but not necessarily the same bytes that the validating core has read. The core handles a mismatch by checking that the line data matches the line data values from the LLC (for only the read-only bytes in the line) and that no write–read conflict exists (i.e., no locally read byte has its write access information set in the LLC); if not, the core generates a consistency exception. The core updates the line's version to the new value, so that the algorithm —which must validate all lines version mismatches— can eventually complete without a mismatch. Although a core might repeatedly retry read validation, RCC is livelock and deadlock free because a version mismatch means that some other core made progress by writing to the LLC.

(3) Post-commit: The core clears its private access information, and the LLC clears all of its access information for the core. By dividing committing of writes into pre- and post-commit, RCC ensures that commit and validation appear to happen together atomically.

In addition, the core must *invalidate all lines* in its private cache, in order to ensure coherence at SFR boundaries. This *self-invalidation* step degrades locality, increasing time

Algorithm 9 A core performs read validation

```
1:  repeat
2:    mustRevalidate ← false
3:    for all private cache lines L with a read-only byte do
4:      let v ← getVersion(L)
5:      Send L's address and v to LLC
6:      resp ← LLC's response                          ▷ resp is ⊥ or ⟨v′, w′, d′⟩
7:      if resp ≠ ⊥ then
8:        ⟨v′, w′, d′⟩ ← resp                          ▷ LLC line's version, write bits, & data values
9:        mustRevalidate ← true
10:       d ← getData(L)
11:       if d′ ≠ d ∨                                  ▷ Compares read-only bytes only
              w′ ∩ getReadBits(L) ≠ ∅ then
12:         Consistency exception!
13:       end if
14:       setVersion(L, v′)
15:     end if
16:   end for
17: until not mustRevalidate
```

and communication costs, especially for short regions. Section 4.5 introduces optimizations that avoid the costs of self-invalidation.

Write-after-read upgrades. RCC's use of value validation requires careful handling of *write-after-read (WAR) upgrades*. A WAR upgrade happens when a core writes a byte that it read earlier in its ongoing region. Simply overwriting the byte in the private cache line would make it impossible to value-validate read(s) performed earlier in the region. RCC thus sends an upgraded line's read access information and version to the LLC. The LLC immediately read-validates the line (similar to its handling of private cache line evictions), and it detects future read–write conflicts for the line. As Section 4.4.1 describes, the RCC architecture avoids the cost of communication with the LLC on upgrades to L1 lines, by relying on the read access information in the L2's corresponding line.

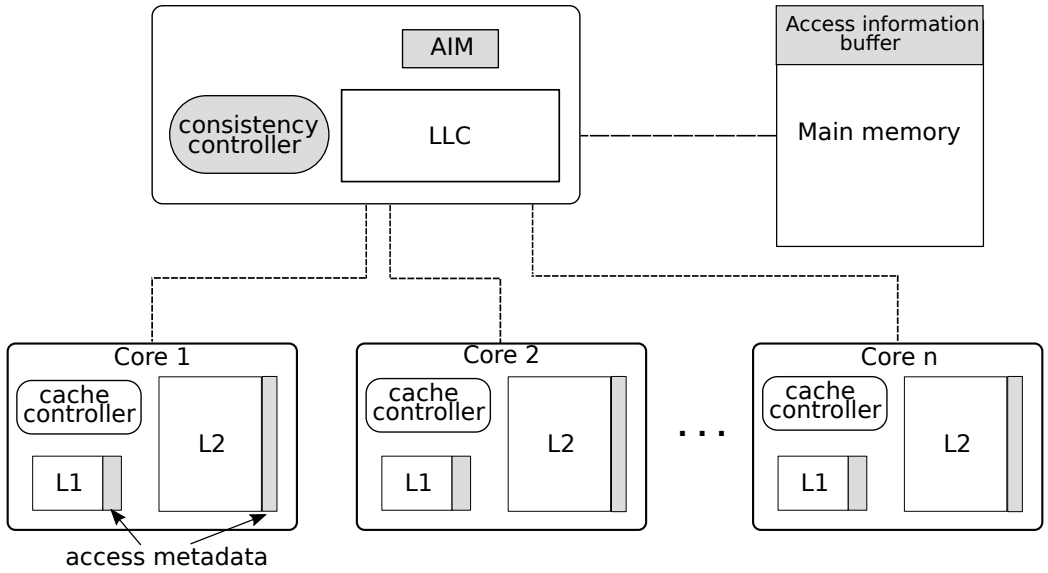


Figure 4.2: The RCC architecture (not according to scale). The shaded parts show additional hardware structures introduced in the design.

4.4 Architecture of RCC

The RCC architecture is a collection of modifications to a commodity multi-core processor. We assume that each core has a cache hierarchy with private, write-back L1 and L2 caches, and that cores share the last-level cache (LLC). This section describes modifications to a base architecture that has *no support for cache coherence*. In the base architecture, each cache line has only a *valid* bit and a *dirty* bit. The private caches are inclusive and the LLC is not inclusive. Figure 4.2 shows the components that RCC adds to the processor: (1) access information storage and management and (2) a consistency controller for ensuring consistency at the LLC.

4.4.1 Private Access Information Management

Each core maintains access information for each of the lines in its private caches. RCC associates two bits per byte with each line in the core's L1 and L2 cache, as shown in Figure 4.3. One bit is the byte's *read bit* and the other is the byte's *write bit*. A byte's *read bit* indicates that the byte was read, without first being written, during an SFR. A byte's *write bit* indicates that the byte was written (and potentially subsequently read) during an SFR. Both bits are set if and only if an SFR reads and then writes the byte.

Updating access information. When a core writes a byte, it sets the byte's write bit if it is not already set. When a core reads a byte, it sets the byte's read bit only if byte has not been *read or written*.

When a core writes a byte in the L1 that was previously read in the same region, the core experiences a *write-after-read (WAR) upgrade*. RCC handles WAR upgrades specially to allow for correct value validation. Simply overwriting the byte in the L1 would make it impossible to validate the values of reads performed earlier in the region, before the WAR upgrade. To make value validation work, RCC writes the line's access information to the L2, before letting the write execute. When a core evicts a dirty L1 line to the L2, RCC checks the L2 line's access information. If the L2 line was previously read, RCC immediately validates reads to the line using the mechanism described in Section 4.4.3.2. After validating reads, RCC writes back the dirty L1 line and its access information to the L2.

Evictions. When a core evicts a line from the L1 to the L2, the line's access information is copied to an identical bit array for the line in the L2. When the L2 evicts a line, it sends

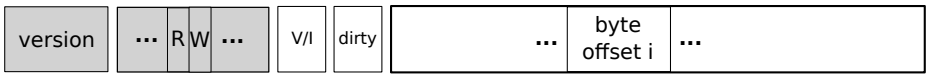


Figure 4.3: Per-line metadata introduced by RCC for private caches. Metadata added by RCC is shaded gray.

the line's access information to the *access information memory* (AIM), which is co-located with the LLC; Section 4.4.2 describes the operation of the AIM and the LLC.

Versions. In addition to read and write bits, each cache line in the L1 and L2 has an associated *version*. The version is a 32-bit field that RCC uses to detect consistency violations (Section 4.3.3).

4.4.2 LLC Access Information Management

Rather than storing access information for each LLC line in the LLC or in memory, RCC stores the information in the *access information memory* (AIM). The AIM is a cache-like memory connected on a bus shared with the LLC. Each AIM entry contains two bits per byte of access information for each core in the system. An AIM entry also contains a 32-bit version, used during read validation. For a system with C cores and B-byte cache lines, the size of an AIM entry is $2 \times C \times B + 32$ bits; 1,056 bits per entry for a typical 8-core system with 64-byte cache lines. Figure 4.4 illustrates the structure of an AIM entry.

When a core writes back a line to the LLC, the LLC updates the line's AIM entry to reflect the access information metadata in the line being written back. The LLC copies the core's updated access information from the private line's metadata into the core's access information bits in the line's AIM entry. When a core writes back a dirty line to the LLC,

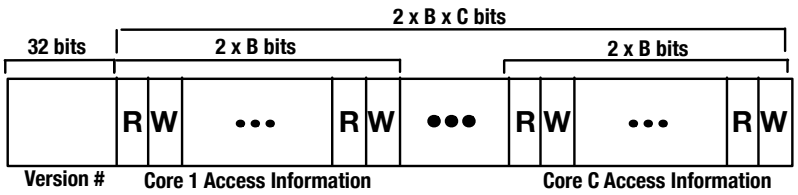


Figure 4.4: An AIM entry for a system with C cores and B-byte cache lines.

the LLC also increments the version for the line that is stored in the AIM. Note that only the LLC, not the private caches, updates a line's version.

In an idealized RCC implementation, the AIM would contain one entry for every line in the LLC, with the same associativity. Evicting AIM lines only when the LLC evicts corresponding lines makes such an idealized AIM a perfect cache of the LLC's access information. However, an ideal AIM is impractical: in a system with 8 cores, 64 byte LLC lines, and a 16MB LLC, the AIM would be around 33MB—an impractically large (88 mm^2), slow (5.3ns), and power hungry (7W leakage) on-chip structure in 32-nm technology (data from CACTI 5.3 [103]). Therefore for an 8-core system, the RCC design considers a realistic AIM that has 32K entries ($\sim$4MB) and only 4-way associativity, rather than the LLC's 16-way. A 4MB AIM is reasonably implementable, with lower area (11 mm^2), latency (1.9ns), and leakage power (900mW). In a high-end, 32-nm Intel Core i7-3970X[17] at 3.5GHz, the AIM would impose a reasonable 2.5% area overhead, 7-cycle access latency (easily hidden by LLC latency), and leakage at a tolerable 0.6% of TDP [95].

The size of an entry in the AIM scales with the number of cores. At realistic CMP core counts, the AIM's hardware cost is not prohibitive. At 16 cores, a 32K-entry AIM is realizable with a 2.6ns access time, 26 mm^2 area overhead, and 1.8W leakage power. At 32

[17] `http://ark.intel.com/products/70845`

cores, a 16MB AIM with 32K entries is costly, but realizable with a 3.9ns access time, 64 mm^2 area cost, and 3.8W leakage power. A less costly 16K-entry AIM for a 32 core machine has a latency of 3.9ns, area of 45 mm^2, and leakage power of 2.2W. As Section 4.6.4 shows, the AIM remains effective across core counts; for 32 cores, the smaller 16K entry AIM design remains effective.

While the original CE algorithm [115] backed metadata directly to memory, we observe that adding an AIM-like metadata cache may reduce CE's traffic to memory, as it reduces RCC's. However, this observation reveals a fundamental distinction between RCC and CE. Unlike RCC, CE assumes a coherence mechanism (e.g., MOESI) that eagerly exchanges access information bits with coherence messages. Consequently, CE with an AIM would incur the AIM's hardware cost and complexity *on top of* a full MOESI coherence protocol implementation. Furthermore, an implementor of CE with an AIM must design, implement, and verify the AIM, the coherence protocol, *and* their interactions via the conflict detection mechanism. RCC only needs an AIM, and its conflict detection mechanism, making its hardware cost and complexity lower than that of CE.

Virtualizing access information to memory. Regardless of the geometry of the AIM (ideal vs. cache-like), RCC must preserve the access information in an AIM entry in memory when the entry is evicted from the AIM. A line is evicted from an ideal AIM exactly when a line is evicted from the LLC, and from a cache-like AIM at least as often. Similar to prior work [115], RCC maps evicted AIM entries into a dedicated region in memory.

To enable post-commit to clear all access information for a core without explicitly tracking and updating access information that has been evicted to memory, RCC augments an evicted AIM entry with a list of saved *epochs*, one per core, before pushing the entry to

memory. An epoch is a number that identifies a core's SFR. The AIM maintains an epoch for each core in a *current epoch register*. A core's current epoch register in the AIM is incremented whenever a core finishes an SFR. When the AIM fills a line, it compares the incoming entry's saved epochs to each core's current epoch register. If the epochs differ, the AIM clears the access information for that line for that core. It is then correct to clear a core's access information because the saved epoch indicates that the access information represents accesses from a previous SFR.

Storing access information in memory. For a system with C cores, B-byte cache lines, V-bit versions, and E-bit epochs, RCC must preserve $P = C \times (E+2) + V/B$ bits per byte of access information from the AIM. RCC reserves the high-order $\arg\min_i(2^i \geq P/8)$ address bits and uses addresses with those bits set to store evicted access information. With 8 cores, 64-byte lines, 32-bit versions, and 32-bit epochs, a system needs $P = 272.5$ bits per byte of backing memory. In such a system, RCC *reserves* (i.e., does not *allocate*) 9 address bits, leaving the application with a 55-bit address space. The AIM computes an entry's location in memory using the 55-bit line address as the entry's memory location's high-order bits.

Note that while these addresses must be reserved for correctness, they are *not* necessarily a source of memory overhead. AIM evictions are infrequent, and these reserved memory locations are unlikely to occupy any physical memory, in a way similar to page tables. Consequently, the RCC's translation scheme is unlikely to present a serious performance or memory capacity problem. Moreover, Section 4.6 shows experimentally that AIM evictions are usually rare.

4.4.3 Consistency Controller (CC)

RCC ensures consistency using a *region commit protocol* that is implemented in RCC's *consistency controller* (CC). Section 4.3.3 described the basic operation of the region commit protocol. Here we focus on the CC's implementation.

The CC is co-located with the AIM, and has several responsibilities: comparing access bits from a single core with access bits from other cores, checking cache line versions during read validation, and coordinating with RCC's core logic to initiate value checking when a version check fails.

4.4.3.1 Region Commit Protocol

The CC has fast read/write access to the AIM via a bus shared by the CC, LLC, and AIM. The CC is connected to each core, enabling cores to send lines to the CC for conflict checking. When a core reaches a region boundary, it works with the CC to detect conflicts and validate reads.

Pre-commit. During pre-commit, the core streams access information from its dirty, privately cached lines to the CC. The CC buffers the lines' write access information while it reads in the line's access information from the AIM. The CC then compares the validating core's access bits to all other cores' access bits using fixed-function combinational logic. The logic uses a single multiplexer to select the core's access bits and then computes a bitwise `and` of those bits with all other cores' bits. If a logical `or` of the bits in the result is nonzero, then the bits indicate a conflict and the CC delivers an exception. If not, the CC updates the access bits in the AIM to match the buffered ones it received from the core.

Read validation. After pre-commit, the CC begins read validation. The core streams a sequence of messages to the CC, one for each line the core read during the ending region. Each message contains the line's address and version from the core's private cache. The CC fetches addresses and versions from the AIM for each message it receives from the core. The CC uses dedicated logic to compare the line's version in the core's message to the version from the AIM. If all versions match and no write bits are set for a remote core for any offset in the shared line, read validation completes successfully. If a read line's versions match, but a write bit was set by a remote core, the CC responds with read bits and checks for write–read conflicts. In case of a conflict, the core raises a consistency exception.

If a line's version differs, then another core wrote the line during the ending region and there may be a conflict. On a version mismatch, the CC messages the core with the line's address and updated version. The core re-fetches that line from the LLC into a dedicated line comparison buffer in the core. The core compares the (read-only) line in the private cache to the line in the comparison buffer.

If the lines differ, then the validating core read inconsistent data and raises a consistency exception. If they match, then the core may have seen consistent data in its region. On receiving a version mismatch message from the CC, the core also sets its *revalidate bit*. The revalidate bit indicates that after the core finishes validating all remaining lines, it must start again from the beginning, streaming version messages to the CC, to ensure that it saw consistent data. After the core completes validation without version mismatches, it unsets the revalidate bit and continues.

Post-commit. During post-commit, a core prepares for its next region. The core streams dirty bytes in its L1 and L2 caches to the LLC, and it clears its L1 and L2 cache's access

information (e.g., using gang clearing [124]). The CC clears the core's access information in the AIM. The AIM clears access information for lines already evicted to memory *lazily*, when the line is next cached in the LLC). Finally, the AIM increments the committing core's epoch and the core continues to the next region.

4.4.3.2 Other CC Responsibilities

Handling evictions to the LLC. When an L2 evicts a line with access information, the CC performs pre-commit and read validation on the line. The CC checks for conflicts using the access information in the AIM, and checks that the L2 line's bytes match the version or values in the LLC. Finally, the CC updates the line's access information in the AIM.

When a core's L2 fetches an LLC line with access bits in the AIM for that core, the LLC sends the core the line's data values and the core's access bits, which the cores uses to populate its L1 and L2 access information. The AIM then stops tracking access bits for the line for that core.

Delivering consistency exceptions. RCC raises a consistency exception whenever the CC or the core detects a conflict. When RCC detects a consistency exception, it raises a dedicated per-core signal for the core that detected the conflict. The consistency exception signal is a non-maskable interrupt. By default, the core that received the interrupt should execute operating system code to terminate the program's execution.

4.5 Design Optimizations

Self-invalidation at region boundaries is a key source of overhead in the RCC design presented in Sections 4.3 and 4.4. It hurts locality *across* region boundaries, leading to degraded performance. The impact of lost temporal locality across region boundaries becomes more

acute for programs that have shorter regions. This section introduces optimizations that focus on reducing *self-invalidations* soundly, to improve locality across region boundaries. We also describe how to reduce traffic generated by region commits.

4.5.1 Avoiding Self-Invalidation

We introduce different optimizations for *touched* (read or written by the current region) and *untouched* lines.

4.5.1.1 Touched Lines

The intuition behind optimizations for touched optimizations is that pre-commit and read validation already process these lines and can check if they are up-to-date and thus do not need to be invalidated. (Note that post-commit always clears all private cache lines' *access information*.)

Read-only lines. RCC need not invalidate privately cached lines that are *read-only*. This optimization is correct because read validation already ensures that read-only lines in the private cache are consistent with the (shared) copy in the LLC.

Dirty lines. For *dirty* lines, pre-commit can check if a line's version is unchanged in the LLC—a sufficient condition for not invalidating the line. This optimization extends pre-commit to send the core's cached version of the line to the CC. If the version matches the value in the AIM, then the core has the latest version and does not need to invalidate the line. On a version mismatch, the CC sends a message asynchronously to the core indicating that it must in fact invalidate the line. If the core receives no message for a dirty line, it does not invalidate the line during post-commit.

4.5.1.2 Untouched Lines

An untouched line need not be invalidated if RCC can ensure that other cores have *not* written to the line during the region's execution. We introduce two optimizations to exploit this idea.

A COND-INVALID state The first optimization adds COND-INVALID as a new state for private cache lines. This state indicates that the line's data is valid only if the LLC's version is unchanged. During post-commit, a core changes each untouched line's state to COND-INVALID, instead of *invalid*.

When a core accesses a line in the COND-INVALID state for the first time in a subsequent region, the L2 cache sends the core's copy of the line's version (but not the data values) to the CC, which compares the version with the AIM's copy of the line's version and replies to the core indicating whether the versions match. If the versions match, the L2 and L1 caches upgrade the line to *valid*. Otherwise, the access is handled as a miss. This optimization reduces on-chip traffic by often sending only a version rather than data values on an L2 cache miss. This optimization incurs latency on an access to a COND-INVALID line, due to a roundtrip exchange with the CC. However, the optimization reduces on-chip traffic compared with a regular L2 cache miss, often sending only a version rather than a line's full data.

Write signatures. Second, RCC minimizes self-invalidations for untouched lines by keeping a per-core write signature [43] in the CC that encodes *which* lines have been updated in the LLC during each core's current region *by any other core*. During post-commit, if a line is not in a core's write signature, the core need not invalidate the line.

The CC encodes a write signature for each core's ongoing region as a Bloom filter [24,43]. Whenever *any* core *C* writes back to the LLC, the CC updates *every* core's write signature except *C*'s to include the updated line. When a core starts read validation, the CC sends the core its write signature and clears the copy of the signature in the CC. The core uses its received copy of the write signature during post-commit to identify untouched lines in its private caches. If the signature does not contain the line, then it was definitely not updated in the LLC during the core's execution and it can stay in the *valid* state in the core's private caches.

RCC uses a small, 112-bit Bloom filter for each core which, along with control data, fits into one 16-byte network flit. We use two hash functions that each set a Bloom filter bit. A small Bloom filter is sufficient to encode a write signature for short regions. Catering to short regions pays off because they suffer self-invalidations most frequently.

4.5.2 Optimizing Region Commit

The following optimizations minimize the work performed by RCC at region boundaries.

Optimizing read validation. Our insight for optimizing read validation is that a core *C* can forgo validating a line if the line was not updated in the LLC by any other core during *C*'s region. To do this check, *C* uses the per-core write signature introduced in Section 4.5.1, obtained before read validation starts. To ensure atomicity, *C* re-fetches the write signature *after* read validation to ensure it has not changed (if it has, *C* restarts read validation).

Deferring write-backs. In the base RCC design, pre-commit writes back both write access information and data values for dirty cache lines. We optimize pre-commit by avoiding sending the data values to the LLC until (and if) they are needed by another core.

RCC implements this optimization by adding $\log N$ additional bits (for a system with N cores) to each cache line in the LLC to identify the "last-writer" core that has up-to-date data, plus an additional bit to indicate whether the line's state is "deferred." If another core requests a deferred line from the LLC, the LLC first fetches the latest values from last-writer core. While this optimization is analogous to the Owner state in the MOESI protocol [172], RCC derives greater benefit from it by avoiding otherwise-mandatory write-backs at every region boundary.

We find that deferring write backs at the granularity of individual cache line offsets lead to muted additional benefit at a greater hardware complexity.

4.6 Evaluation

This section evaluates the performance and on-chip network and off-chip memory traffic of RCC and compares with competing approaches.

4.6.1 Simulation Methodology

We have implemented RCC in simulators based on the RADISH simulator provided by its authors [54]. For comparison, we have implemented a directory-based MESI cache coherence protocol [172] to model current shared-memory systems, which we call *MESI*. We have also implemented *Conflict Exceptions* (CE) [115] on top of MESI. The simulators consume a serialized trace of events generated by a Pintool [117]. Multiple simulator configurations process the *same* trace, in order to eliminate differences due to run-to-run nondeterminism. All three simulators model a realistic baseline architecture, detailed in Table 4.1. The MESI simulator's LLC is inclusive in order to support a directory protocol [172]. We model a directory embedded in the LLC with the same associativity as the LLC. The MESI protocol (see Figure 8.6 in [172]) performs silent evictions from *E*

Processor	8-, 16-, or 32-core chip at 1.6 GHz. Each non-memory-access instruction takes 1 cycle.
L1 cache	8-way 32 KB per-core private cache, 64 B line size, 1-cycle hit latency
L2 cache	8-way 256 KB per-core private cache, 64 B line size, 10-cycle hit latency
Remote cache hit	15-cycle one-way cost
LLC	64 B line size, 35-cycle hit latency
8 cores:	16-way 16 MB shared cache
16 cores:	16-way 32 MB shared cache
32 cores:	32-way 64 MB shared cache
AIM cache	4-way metadata cache with 32K lines
8 cores:	132 B line size ($\sim$4 MB), 4-cycle hit latency
16 cores:	260 B line size ($\sim$8 MB), 5-cycle hit latency
32 cores:	516 B line size ($\sim$16 MB), 7-cycle hit latency
Memory	120-cycle latency
Bandwidth	NoC: 100 GB/s, 16-byte flits; Memory: 48 GB/s

Table 4.1: Architectural parameters used for simulation.

to *I*, but evictions of shared cache lines (*S* to *I*) send a message to the directory. The CE simulator extends the MESI simulation to eagerly detect conflicts [115]. The CE algorithm requires memory access on private cache evictions to back up access metadata and to fetch access metadata on LLC hits under certain conditions [115]; our CE simulator optimistically assumes that the latency of accessing memory is masked by subsequent memory operations. The RCC simulator's LLC is *not* inclusive (Section 4.4).

We evaluate the scalability of the three simulators with a varied number of core counts. By default, the RCC simulator models a realistic AIM cache, as Table 4.1 shows.

Estimating execution time. Table 4.1 shows the number of cycles required for memory and non-memory instructions. All three simulators report the maximum number of cycles for any core; as in prior work [17,54], cores do not model time spent waiting in synchronization. All three simulators model wait-free write-back caches with idealized write buffers.

The RCC simulator models the costs of RCC performing operations at region boundaries. Since cores send multiple messages without waiting synchronously for responses during the pre-commit and read validation phases, we compute the cycle cost of messaging based on the total size of messages sent and the available bandwidth between a core and the LLC. During read validation, a core sends lines' versions to the LLC. Each 16-byte flit contains four lines to be validated, since a flit can fit four tags and versions plus a control block. We assume that the CC and LLC are ported to handle a flit's four validation requests at a time. The RCC simulator models version mismatches, including the costs of the CC alerting the core and the core restarting read validation. The RCC simulator models post-commit, including gang-clearing for self-invalidation of private cache lines and bulk-clears of per-core AIM information.

Estimating network traffic. We simulate an on-chip network and off-chip memory network with 16-byte flits and the bandwidth characteristics shown in Table 4.1. Control messages are 8 bytes (tag plus message type); a MESI data message is 64 bytes (corresponding to a cache line). For RCC write-backs we model idealized write-buffer coalescing that sends only the dirty bytes in a line.

Benchmarks. Our experiments execute the PARSEC benchmarks [17], version 3.0-beta-20150206, with simmedium inputs. We include 11 of 13 benchmarks; freqmine uses OpenMP as the parallelization model, and facesim does not finish executing with our Pintool and simulators. We report cycles and traffic for the parallel "region of interest" (ROI) only [16]; vips lacks an ROI annotation so we consider its entire execution to be the ROI. We *include* the costs of stack-local accesses in both simulators, although Pin can identify them.

	Threads	Average accesses per SFR		
		n = 8	n = 16	n = 32
blackscholes	$1+n$	9,150,000	4,570,000	2,290,000
bodytrack	$2+n$	63,600	57,400	47,800
canneal	$1+n$	5,470,000	2,746,000	1,370,000
dedup	$3+3n$	36,300	36,300	35,900
ferret	$3+4n$	630,000	514,000	388,000
fluidanimate	$1+n$	131	99	68
raytrace	$1+n$	5,820,000	3,030,000	1,550,000
streamcluster	$1+2n$	4,320	2,260	1,250
swaptions	$1+n$	83,000,000	41,600,000	20,800,000
vips	$3+n$	105,000	81,100	55,800
x264	$1+2f$	208,000	202,000	202,000

Table 4.2: Threads spawned and average region sizes (rounded to 3 significant figures) for the PARSEC benchmarks. n is the minimum threads parameter in PARSEC. f is the input-size-dependent number of frames processed by x264.

Table 4.2 shows how many threads each benchmark spawns, parameterized by n, which is PARSEC's *minimum threads* parameter (the -n flag). The simulators set n equal to the number of cores in the simulated architecture, which is either 8, 16, or 32 in our experiments. The simulators map thread identifiers to cores using modulo arithmetic. The last three columns show the average number of memory accesses performed per SFR for n=8, n=16, and n=32.

4.6.2 Run-Time Performance and Traffic

Figures 4.5 and 4.6 show our main results. The data show that the execution time and on-chip network traffic of MESI, CE, and RCC are comparable, often significantly favoring RCC, and that these numbers scale with increased core count. The data also show that RCC's off-chip memory traffic is comparable to MESI and uniformly better than CE, in some cases by an order of magnitude or more. The figures group together configurations with the same

core count. The first three bars, *MESI-8*, *CE-8*, and *RCC-8*, show the performance and traffic overheads running on 8 cores. Correspondingly, the next two groups of three bars report the overheads on 16 and 32 cores. All bars are normalized to *MESI-8*.

Overall, the data show that the execution time and on-chip network traffic of MESI, CE, and RCC are comparable, and that these numbers scale with increased core count. The data also show that RCC's off-chip memory traffic is comparable to MESI and uniformly better than CE, in some cases by an order of magnitude or more.

4.6.2.1 Performance

Figure 4.5(a) shows executed cycles as reported by the simulators. Each bar shows the breakdown of execution cycles into different components. *MESI* and *CE* (which builds on MESI) are divided into cycles attributed to *coherence* and *other execution*. Coherence cycles are those spent when the directory forwards requests to remote cores and for core-to-core communication. For RCC, cycles are divided into cycles for pre-commit and read validation and cycles incurred during region execution.

Figure 4.5(a) shows that CE adds minimal performance overhead over MESI (e.g., 0.4% for 8 cores), because our CE simulator does not ascribe any additional cost for transmitting access metadata piggybacked on MESI coherence messages or to access memory or to back up or fetch access metadata; hence the performance of CE is similar to MESI.

The figure shows that RCC outperforms MESI and CE in several cases by avoiding the latency of MESI coherence. RCC underperforms MESI and CE in a few cases (canneal and fluidanimate). For fluidanimate, regions are short and incur latency from cache misses due to frequent self-invalidation. (Although read validation and pre-commit perform substantial work, they incur low latency since they are streaming operations.) For canneal, RCC incurs many AIM cache misses while fetching access metadata for LLC hits. This results in

fetching the access metadata from memory, in effect incurring the latency of a memory access. On average, RCC performs similarly to MESI and CE, outperforming them by 3–4% with 8 cores and performing nearly identically with 16–32 cores.

4.6.2.2 On-Chip Traffic

Figure 4.5(b) compares the on-chip network traffic incurred by MESI, CE, and RCC, counted in 16-byte flits and normalized to MESI with 8 cores. On-chip traffic for RCC includes all communication between cores and the LLC/CC. On-chip traffic for MESI and CE includes all traffic between cores and the LLC, as well as core-to-core communication. The figure uses the same simulator configurations and the same breakdowns for these configurations as in Figure 4.5(a).

As described in prior work [115] and emulated in our simulator, the CE protocol piggybacks on MESI coherence messages and transfers access bits to detect region conflicts. This results in CE incurring more on-chip network traffic than MESI. In addition, CE scales poorly with fluidanimate, since CE must perform many broadcasts of access metadata (endR messages) at fluidanimate's frequent region boundaries [115].

The key result from Figure 4.5(b) is that for all benchmarks except fluidanimate, RCC's traffic overhead increases proportionately with core count, and RCC's traffic scalability is nearly identical to MESI's. This result shows that RCC's traffic overhead is unlikely to prevent scaling to moderate core counts. The reason for the disproportionate increase in fluidanimate's traffic is that fluidanimate performs more writes and synchronization operations with increasing numbers of threads. With 16 worker threads, fluidanimate has 17% more writes and 44% more synchronization operations compared to 8 worker threads, while it has 57% more writes and 146% more synchronization operations with 32 threads. Thus, the benchmark has progressively smaller regions with more threads (Table 4.2). More

frequent region boundaries cause more frequent pre-commit, read validation, and self-invalidation, which all increase traffic. This explanation is clear in the increased proportions of RCC's pre-commit and read validation costs for fluidanimate with 16 and 32 cores.

RCC outperforms MESI and CE for swaptions with 16 and 32 cores. Swaptions has the largest SFRs among the PARSEC benchmarks, and consequently large working sets. With 16 and 32 cores, the number of private and shared cache misses increase significantly for swaptions because of inclusivity, the private caches and the LLC are inclusive in MESI and CE. Eviction of shared lines from the LLC in MESI requires recalling the lines from all private caches, incurring high traffic for shared lines.

On average, RCC adds slightly more on-chip network traffic than MESI but less than CE. Is the *raw magnitude* of this traffic a cause for concern? The *On-chip* columns in Table 4.3 show the average on-chip bandwidth used, in GB/s, for MESI, CE, and RCC on 32 cores. For fluidanimate, the CE algorithm incurs high on-chip network traffic (84 GB/s) and *almost* saturates the on-chip network. For canneal and streamcluster, RCC's algorithm for ensuring consistency, which defers coherence, stresses the network much less compared to eager invalidation-based protocols such as MESI. Even in cases where RCC adds more on-chip traffic than MESI, the average bandwidth used is significantly lower than the available bandwidth (Table 4.1).

4.6.2.3 Off-Chip (LLC-to-Memory) Traffic

Figure 4.6 shows the LLC-to-memory (off-chip) traffic for MESI, CE, and RCC. CE incurs high off-chip traffic overhead over MESI, since the design backs up and fetches evicted access metadata information to and from memory [115]. In particular, CE must back up access bits in a *global* table when a line that was accessed in an ongoing region is evicted from a private cache or the shared LLC. Similarly, the CE design requires memory

	On-chip			LLC-to-memory		
	MESI	CE	RCC	MESI	CE	RCC
blackscholes	<0.1	0.1	<0.1	<0.1	<0.1	<0.1
bodytrack	1.3	1.4	2.4	<0.1	0.3	<0.1
canneal	61	66	37	2	116	9
dedup	4	4	5	1	2	1
ferret	4	4	4	<1	2	<1
fluidanimate	8	84	46	<1	1	<1
raytrace	1.8	1.8	1.5	0.3	0.8	0.3
streamcluster	34	40	16	<1	48	<1
swaptions	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
vips	9	9	9	1	4	1
x264	4	4	4	<1	2	<1

Table 4.3: Average on-chip and off-chip (LLC-to-memory) bandwidth (rounded to one place after decimal) required by MESI, CE, and RCC for 32 cores. For the benchmarks not shown, the maximum value in any column is $\leq$2.2 GB/s.

traffic even on an LLC hit, if the line was evicted to memory or the core has evicted a line from its private caches during the ongoing region. The benchmark streamcluster has the highest proportion of LLC hits that require CE to fetch metadata from memory, leading CE to incur very high memory traffic overhead (510–3,500X) compared to MESI. The absolute bandwidth required by CE for streamcluster with 32 cores is 48 GB/s, as Table 4.3's *Off-chip* columns show, which saturates the memory network capacity. For canneal, the required bandwidth saturates the memory network at 32 cores and exceeds the assumed NOC bandwidth (Tables 4.1 and 4.3), rendering the CE design unimplementable.

The memory requirement for RCC is comparable to MESI for all benchmarks except canneal. The increased traffic for canneal is due to a bounded AIM cache, which leads to misses in metadata lookups. Despite the *relative* increase in memory traffic, *RCC* incurs reasonable average memory traffic for canneal up to 32 cores (9 GB/s as Table 4.3 shows)—a

fraction of the the 48 GB/s maximum available bandwidth (Table 4.1)—and $\leq$2 GB/s for all other programs.

4.6.3 Impact of Optimizations

Our default RCC configuration includes all optimizations presented in Section 4.5. This section evaluates the effect of those optimizations by evaluating RCC without optimizations. We focus on on-chip network traffic since the optimizations affect that metric the most. Figure 4.7 shows the on-chip network traffic incurred by MESI and different configurations of RCC, for 8 cores only (for brevity), normalized to MESI. *RCC unopt* includes none of Section 4.5's optimizations; it incurs 63% more traffic than MESI on average. For fluidanimate, *RCC unopt* incurs high on-chip traffic relative to MESI; the figure shows fluidanimate using a separate y-axis scale. *RCC inv opt* uses only the optimizations for reducing self-invalidations (Section 4.5.1), thereby reducing traffic substantially for fluidanimate and other programs. On average, *RCC inv opt* incurs 23% more traffic than MESI. Even after the self-invalidation optimizations, *RCC inv opt* continues to incur substantial on-chip traffic relative to MESI, due to several sources of traffic incurred by RCC but not MESI. As the figure shows, for all of these programs except swaptions, RCC's higher traffic is due to pre-commit and read validation, which send dirty lines and read validation information, respectively, to the LLC at every region boundary. For swaptions, RCC's higher traffic is due to region execution—the traffic is due to evictions of read-only or upgraded lines from private caches to the LLC, which incur higher traffic costs (to transfer read access information) in RCC than MESI. The last configuration, (fully optimized) *RCC*, reduces traffic further by optimizing the data transmitted during pre-commit and read validation, incurring only 5.7% average traffic over MESI for 8 cores (the same result as in Figure 4.5(b)).

4.6.4 Sensitivity to AIM Cache Size

Our experiments use an AIM cache with 32K entries (Table 4.1). The AIM cache's dimensions increase linearly with the core count, which implies a steeper increase in area, latency, and power consumption. We have thus evaluated the sensitivity of the RCC protocol with both a smaller AIM cache with 16K entries as well as an RCC system with an ideal AIM cache. The top half of Table 4.4 shows the change in run-time performance and off-chip traffic (AIM cache size does not affect on-chip traffic) with an AIM cache with only 16K lines. We find that for 8–32 cores, a 16K-entry AIM increases execution time and on-chip network traffic by less than 1% on average relative to the default 32K-entry AIM. The 16K-entry AIM increases LLC-to-memory traffic on average by less than 7% for 8, 16, and 32 cores.

To evaluate the performance cost of using a small, realistic AIM (the default with 32K entries), the bottom half of Table 4.4 simulates an *idealized* AIM that has one entry for each LLC line. A RCC system with an ideal AIM outperforms a 32K-entry AIM by less than 4% on the average for 8, 16, and 32 cores. Default RCC's performance matches the ideal for all benchmarks except canneal, which has 25–36% run-time overhead for 8–32 cores (result not shown), due to large regions with many AIM accesses that are better handled by the idealized AIM cache. The impact on off-chip traffic due to a bounded-size AIM is limited to less than 18% for 8, 16, and 32 cores. These results show that the RCC protocol remains largely unaffected with a reasonably sized AIM cache.

4.6.5 Comparison with TCC

Although TCC targets speculative execution of programmer-specified regions [85] (Section 6), one could imagine applying its mechanisms to RCC's context. We have

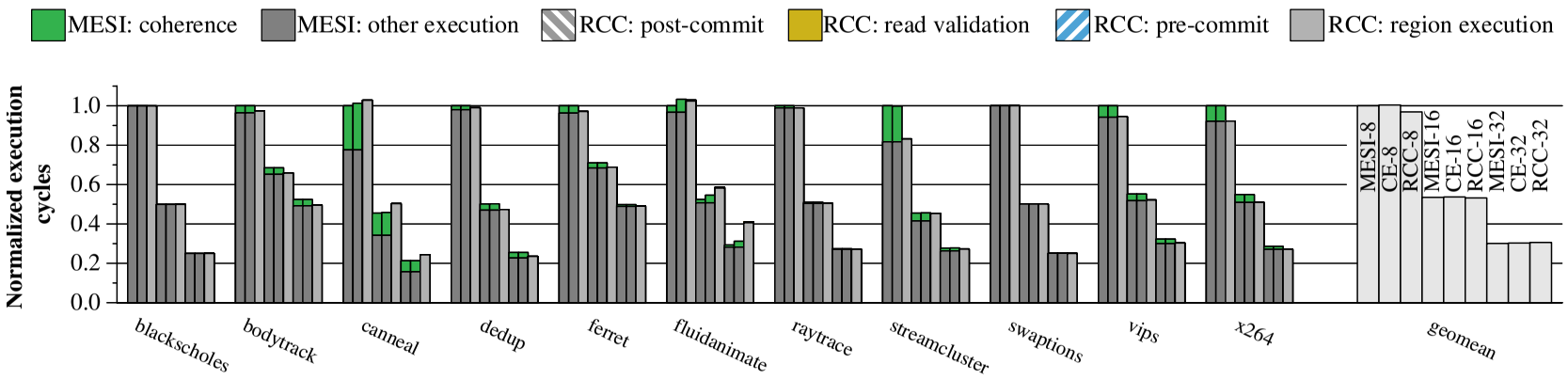
AIM size	Cores	Exec. cycles	Off-chip traffic
16K entries	8	+0.7%	+6.2%
	16	+0.7%	+5.8%
	32	+0.8%	+5.6%
Ideal	8	−2.7%	−8.5%
	16	−3.6%	−15.4%
	32	−4%	−17.2%

Table 4.4: Impact of AIM cache size, relative to the default of 32K entries, on performance and off-chip traffic.

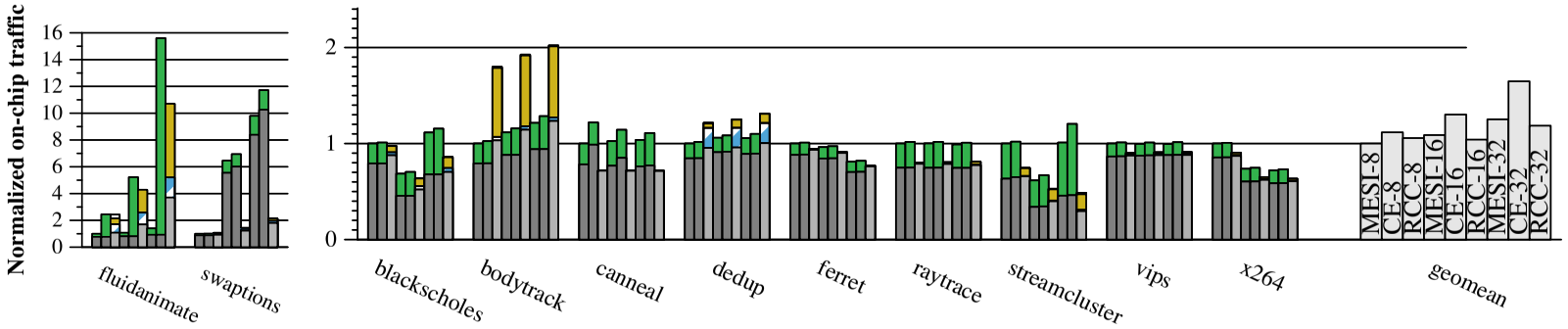
measured the costs that RCC would incur if it used TCC's mechanisms and algorithms. That is, we compute RCC's execution cycles and on-chip traffic *without* pre-commit, read validation, and post-commit, but including the following: each region broadcasts its write set, and a region that overflows its private caches cannot execute in parallel with other overflowed or committing regions [85]. Modeling other costs (e.g., private and shared cache hits and misses) are the same for RCC and TCC. For 8 cores, we find that using TCC's mechanisms increases execution cycles by 3.0X and on-chip traffic by 3.3X, compared with default RCC. For 32 cores, TCC's mechanisms incur an overhead of 5.5X for execution cycles and 9.3X for on-chip traffic compared to RCC. We find that TCC's mechanisms add high on-chip traffic in order to broadcast write sets to all cores, and they incur high run-time overhead because many regions overflow the private caches, leading to much serialization. This comparison shows that, for the same context (precise conflict detection; non-speculative SFRs), RCC's mechanisms provide substantial performance and traffic benefits over TCC's mechanisms.

4.6.6 Summary

Our experiments show that RCC compares favorably to MESI and CE, up to 32 cores, sometimes outperforming both MESI and CE. RCC uses similar on-chip and memory bandwidth to MESI. While CE's on-chip bandwidth is acceptable compared to MESI and RCC, RCC uses far less memory bandwidth than CE, often by orders of magnitude. Our results yield two conclusions. First, RCC provides stronger memory model guarantees than MESI at a similar hardware and run-time cost. Second, we find CE is effectively unimplementable because of its memory bandwidth requirements and, moreover, even if CE used RCC-like metadata caching to decrease memory traffic, CE is fundamentally more complex because it relies on an eager, point-to-point coherence protocol. Finally, the comparison with TCC shows that although RCC's approach is related at a high level to TCC (e.g., marrying coherence and consistency), RCC's mechanism is substantially more practical and implementable than TCC's mechanism. Our evaluation thus shows RCC's value and viability.



(a) Run-time performance.



(b) On-chip network traffic.

Figure 4.5: Run-time performance and on-chip traffic costs for MESI, CE, and RCC for 8–32 cores. The suffix for each simulator indicates the number of cores. The legend at top applies to both graphs.

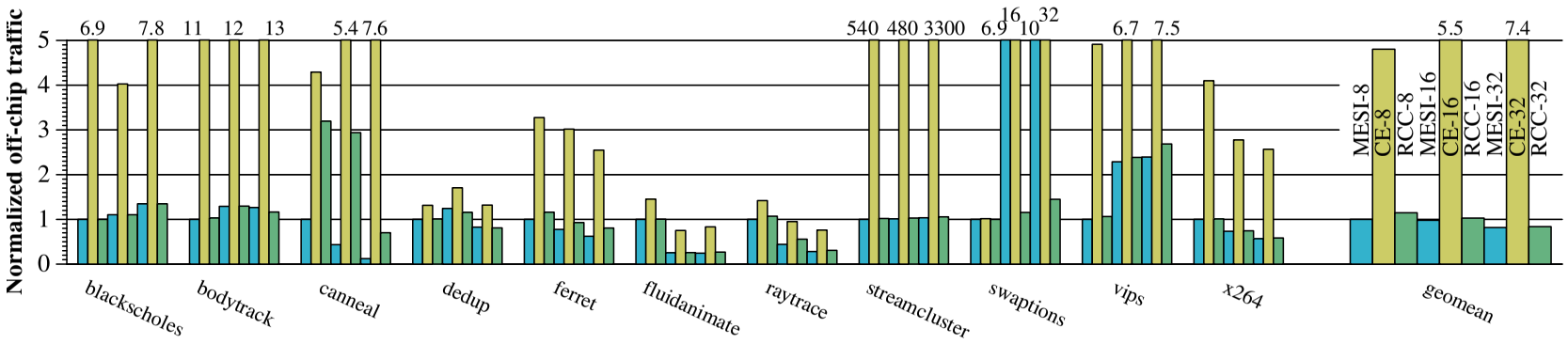


Figure 4.6: LLC-to-memory traffic for for MESI, CE, and RCC for 8–32 cores, using the same configurations as Figure 4.5.

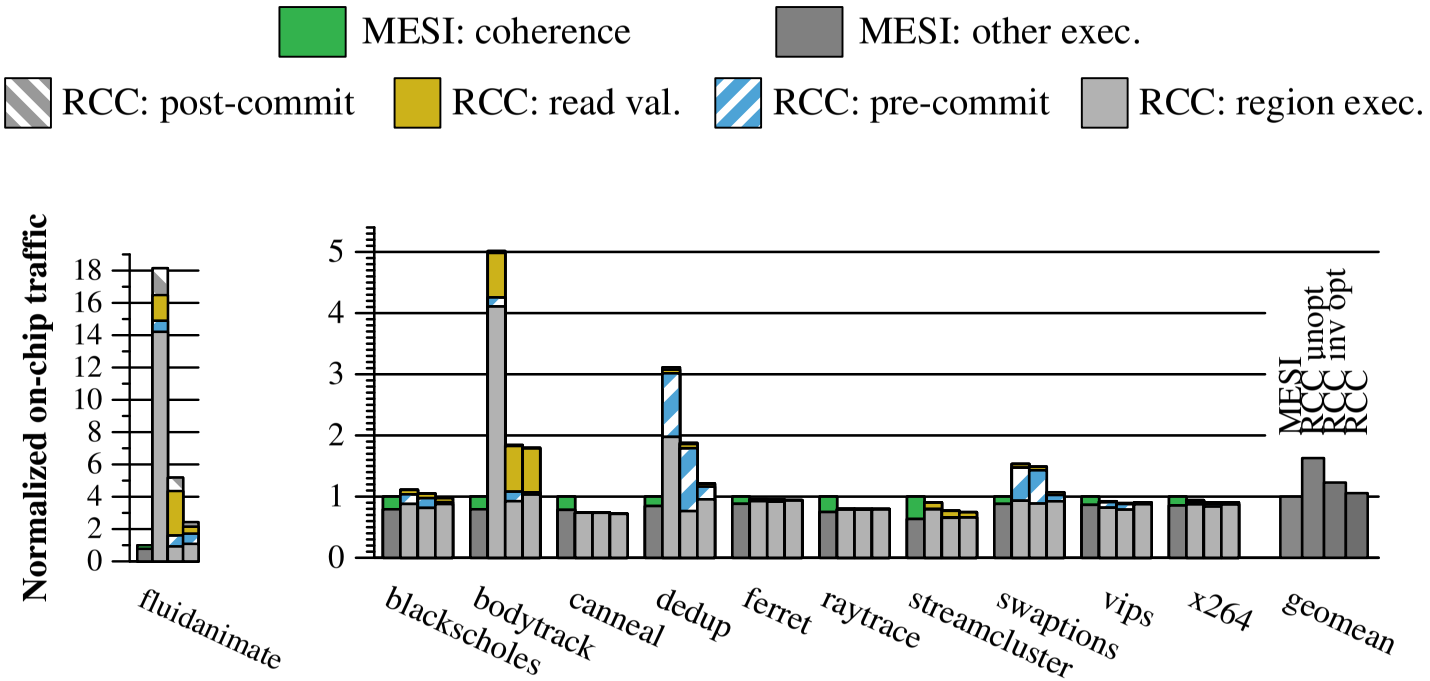


Figure 4.7: The effect of RCC optimizations on-chip network traffic in a system with 8 cores.

4.7 Contributions and Impact

In this chapter, we have presented a novel architectural design called Region Consistency and Coherence (RCC) that provides SFRSx. For every execution, RCC either provides serializability of synchronization-free regions (SFRs), or it generates a consistency exception indicating a data race that may jeopardize consistency. The key to RCC's efficiency is its novel mechanism for detecting region conflicts that allows regions to execute largely in isolation and avoids propagating access information eagerly at the granularity of individual memory accesses. Furthermore, as a result of ensuring consistency, RCC can defer cache coherence until synchronization operations and evictions—unlike existing coherence protocols that ensure coherence at every instruction.

RCC performs competitively with MESI in terms of run-time performance and on-chip network traffic, without incurring CE's high traffic costs [115] or TCC's [85] scalability bottlenecks. For example, RCC significantly outperforms CE in off-chip memory traffic, and we show that the high memory bandwidth required by the CE design effectively makes the algorithm unimplementable in hardware. These results, together with RCC's strong end-to-end guarantees, suggest that RCC significantly advances the state of the art in parallel architecture consistency and coherence.

Despite pervasive cache-coherent architectures, today's systems are both over-designed and under-designed. Coherence protocols are over-designed for common languages [31, 118] that give ill-synchronized code undefined semantics: the coherence protocol wastes resources to ensure that undefined accesses are coherent. Multicore architectures endure high complexity and communication costs in order to maintain the single writer/multiple readers invariant [58,154]. Some transient states exist only to provide coherence semantics to

unsynchronized code [48,172], to which language memory models already ascribe undefined semantics [2, 31, 118].

At the same time, coherent systems are under-designed for the general case of consistency. Systems do not typically ensure consistency, for example, in the presence of operation reordering in a compiler and in hardware write buffers [174]. The result is that programmers must judiciously use synchronization, or endure inscrutable, architecture-specific program behavior. Our proposed work advocates for a departure from cache coherence protocols, instead providing one mechanism that uniformly ensures strong consistency for all accesses (even unsynchronized ones), and permitting aggressive compiler and hardware reordering.

Since RCC provides significantly stronger guarantees than MESI at a comparable cost, it is a compelling design for providing consistency and coherence in future systems. We believe RCC can form the foundation for building efficient shared-memory systems in the future. An architecture like RCC will help programmers build reliable software which can have a long-lasting impact on society.

Chapter 5: DoubleChecker: Efficient Sound and Precise Atomicity Checking

5.1 Introduction

In Chapters 3 and 4, we presented novel techniques to provide serializability of regions that were bounded by synchronization operations. In this chapter, we extend the scope of providing serializability to arbitrary-sized *programmer-defined regions*. For programming language semantics, serializability of arbitrarily-sized regions is also known as *atomicity*: program execution must be equivalent to some serial execution of atomic regions and non-atomic instructions [112]. Atomicity, which is a fundamental non-interference property that eases reasoning about program behavior in multithreaded programs. However, modern general-purpose languages provide crude support for enforcing atomicity—programmers are basically stuck using locks to control how threads' shared-memory accesses can interleave. Programmers try to maximize scalability by minimizing synchronization, often mistakenly writing code that does not correctly enforce needed atomicity properties. Therefore, checking atomicity *is* an important problem. Furthermore, while checking for region conflicts suffice to provide SFR serializability, it is important to *check* serializability of atomic regions rather than conflict freedom since region conflicts are not necessarily errors according to the program semantics.

Motivation. Atomicity violations are common but serious concurrency errors that are sensitive to inputs, environments, and thread interleavings, so violations manifest unexpectedly and only in certain settings. According to a study, 69% of non-deadlock concurrency errors are due to atomicity violations [112]. Checking atomicity in different environments is essential to detect (1) detect and help diagnose atomicity violations that occur only in those execution environments and (2) make software more robust by detecting all atomicity violations that occur, in order to take action such as terminating the program or trying to fix the program automatically [96].

The goal of this work is to reduce the cost of sound and precise atomicity checking significantly in order to increase its practicality for various use cases.

Existing work. Static analysis can check atomicity but is inherently imprecise, and type-based approaches rely on annotations [62, 65, 72, 74]. Moreover, existing static analyses do not scale well to large programs. Dynamic analyses typically check only the current monitored execution, are precise and can scale well to large programs [63, 67, 73, 113, 184, 185, 189]. However, existing dynamic analyses are precise but slow programs by up to an order of magnitude or more. Dynamic approaches incur high costs to track cross-thread dependences, which is especially expensive because it requires inserting intrusive synchronization to ensure correctness. The state-of-the-art *Velodrome* algorithm, which sound and precisely checks *conflict serializability*, slows programs by about an order of magnitude on average [73], mainly because of the high cost of identifying cross-thread data dependences soundly and precisely (Section 2.3.2).

5.2 Design of DoubleChecker

This section describes our dynamic conflict serializability checker, called DoubleChecker [18], that uses two cooperating dynamic analyses to check atomicity without incurring the full costs of tracking cross-thread dependences soundly and precisely for all program accesses. The key insight of DoubleChecker lies in its *dual-analysis* approach that avoids the high costs of precisely tracking cross-thread dependences and performing synchronized metadata updates at every program access, by overapproximating dependences between transactions (a transaction is a dynamically executing atomic region) and then recovering precision only for those transactions that might be involved in violations.

5.2.1 Overview

DoubleChecker achieves low overhead by staging work between two new analyses, one *imprecise* and the other *precise*. DoubleChecker's imprecise analysis, called *imprecise cycle detection* (ICD), monitors all program accesses to track cross-thread dependences soundly but imprecisely, i.e., the dependences imply the execution's actual dependences as well as other false dependences. ICD is inherently imprecise mainly because it identifies dependence edges by tracking shared-memory "ownership"; *a transfer of ownership indicates a possible dependence, but does not guarantee a dependence nor identify the source of the dependence edge*. ICD constructs a *dependence graph* whose nodes are transactions and whose edges correspond to the cross-thread dependences that ICD detects. ICD checks for cycles in this graph—presence of cycles in the dependence graph indicate potential atomicity violations, which are a superset of the true (precise) cycles.

The second analysis, *precise cycle detection* (PCD), is a sound and precise analysis that limits its monitoring to a *subset* of all transactions in the program: the transactions identified

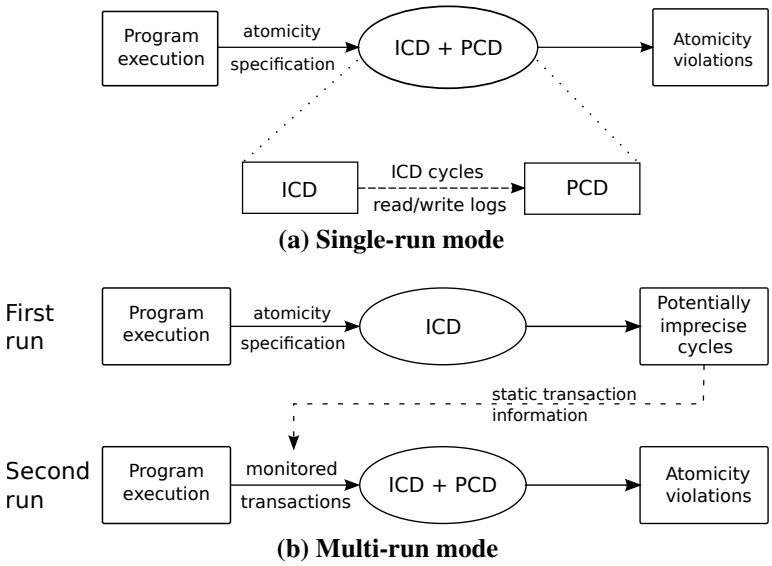


Figure 5.1: An overview of DoubleChecker's two execution modes.

by ICD as being involved in potential cycles—which preserves soundness because every precise cycle's transactions will always be a subset of some (potentially imprecise) cycle identified by ICD. The precise analysis *recomputes* precise cross-thread dependences, and checks for presence of cycles in the precise dependence graph. Note that PCD is *not* a standalone analysis: it performs its analysis on an execution's access log, provided by ICD.

DoubleChecker can operate in either of two execution modes–single-run and multi-run mode. Figure 5.1 overviews the two modes of DoubleChecker.

Single-run mode. In *single-run* mode, ICD and PCD run on the same program execution. ICD logs all program reads and writes and ordering dependences between them, so PCD can identify precise cycles. A key cost of single-run mode is logging all program accesses.

Multi-run mode. In testing and deployment situations, programs are run multiple times with various inputs. DoubleChecker's *multi-run* mode takes advantage of this situation by splitting work across multiple program runs.[18] One run can identify transactions that might be involved in a dependence cycle, and another run can focus its monitoring on this set of transactions. In contrast to single-run mode, multi-run mode avoids logging all accesses during the first run by instead performing precise checking during a second run of the program. The *first* run of multi-run mode uses only ICD. This run identifies all *regular* (non-unary) transactions that are involved in imprecise cycles according to their static starting locations (e.g., method signatures). Rather than identifying precisely which *unary* transactions were involved in cycles, the first run identifies only whether *any* unary transactions were involved in any cycle. It would be expensive to identify unary transactions precisely, since it would essentially require recording the program location of every non-transactional access.

The *second* run takes this static transaction information—set of regular transactions plus a boolean about unary transactions—as input, and limits its analysis to the identified regular transactions and instruments *all* unary transactions if the unary transaction boolean is true. We find this approximation yields acceptable performance in practice since most accesses are *not* unary, i.e., they occur inside regular transactions. In our experiments, the second run uses both ICD and PCD—similar to the single-run mode—for the best performance, but the second run can also potentially use a different precise checker such as Velodrome.

In multi-run mode, DoubleChecker guarantees soundness if the two program runs execute identically. In practice, two executions in the wild will take different inputs and execute different thread interleavings. The set of (static) transactions identified by the first

[18]Prior bug detection work has split work across runs using sampling (e.g., [108]), which is complementary to our work.

run may not be involved in a cycle in the second run; similarly, the second run may execute transactional cycles not present in the first run. To increase efficacy, the second run can take as input all transactions identified across multiple executions of the first run. The multi-run mode can still be effective in practice if the same regions tend to be involved in cycles across multiple runs.

5.3 Imprecise Cycle Detection

Imprecise cycle detection (ICD) is a dynamic analysis that analyzes all program execution in order to detect cycles among transactions. ICD constructs a sound but imprecise graph called the *imprecise dependence graph* (IDG) to model dependences among the transactions in a multithreaded program. The nodes in an IDG are regular transactions (which correspond to atomic regions) or unary transactions (which correspond to single accesses outside of atomic regions). A *cross-thread* edge between two nodes in different threads indicates a (potentially imprecise) cross-thread dependence between the transactions. Two consecutive nodes (i.e., transactions) in the same thread are connected by an intra-thread edge that effectively captures any intra-thread dependences.

We first describe an existing concurrency control mechanism that ICD extends to help detect cross-thread dependences but that makes detection inherently imprecise. We then describe how ICD builds the IDG and detects cycles.

5.3.1 Efficient Tracking of Cross-Thread Dependences

This section describes *Octet*, a recently developed software-based concurrency control mechanism [35] that ICD uses to help detect cross-thread dependences. Octet establishes and identifies *happens-before* relationships [104] that soundly but imprecisely imply all of an execution's cross-thread dependences.

At run time, Octet maintains a locality state for each object[19] that can be one of the following: $\mathsf{WrEx_T}$ (write-exclusive for thread T), $\mathsf{RdEx_T}$ (read-exclusive for thread T), or $\mathsf{RdSh_c}$ (read-shared; we explain the counter c later). Table 5.1 shows the possible state transitions based on an access and the object's current state. To maintain each object's state at run time, the compiler inserts instrumentation called a write barrier[20] before every store:

```
if (obj.state != WrEx_T) { // fast path
  /* slow path: change obj.state */
}
obj.f = ... ;  // program write
```

and a read barrier before every load:

```
if (obj.state != WrEx_T && obj.state != RdEx_T && // fast
!(obj.state == RdSh_c && T.rdShCnt >= c)) {  // path
  /* slow path: change obj.state */
}
... = obj.f;  // program read
```

The state check, called the *fast path*, checks whether the state needs to change (the *Same state* rows in Table 5.1). The key to Octet's performance is that the fast path is simple and performs no writes or synchronization. If the state needs to change, the *slow path* executes in order to change the state.

Conflicting transitions. The last four rows of Table 5.1 show *conflicting* state transitions, which indicate a conflicting access and require a coordination protocol to perform the state change. For example, in Figure 5.2, before thread T2 can change an object o's state from $\mathsf{WrEx_{T1}}$ to $\mathsf{RdEx_{T2}}$, T2 must *coordinate* with T1 to ensure that T1 does not continue accessing o racily without synchronization. As part of this coordination protocol, T1 does

[19] We use the term "object" to refer to any unit of shared memory.

[20] A barrier is instrumentation added to every program load and store [190].

Trans. type	Old state	Access	New state	Cross-thread dependence?
Same state	$\mathsf{WrEx_T}$	R or W by T	Same	No
	$\mathsf{RdEx_T}$	R by T	Same	
	$\mathsf{RdSh_c}$	R by T *	Same	
Upgrading	$\mathsf{RdEx_T}$	W by T	$\mathsf{WrEx_T}$	No
	$\mathsf{RdEx_{T1}}$	R by T2	$\mathsf{RdSh_{gRdShCnt}}$	Possibly
Fence	$\mathsf{RdSh_c}$	R by T *	Same *	Possibly
Conflicting	$\mathsf{WrEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	Possibly
	$\mathsf{WrEx_{T1}}$	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 5.1: Octet state transitions. *A read to a $\mathsf{RdSh_c}$ object by T triggers a fence transition if and only if per-thread counter $\mathsf{T.rdShCnt} < \mathsf{c}$. The fence transition updates T.rdShCnt to c.

not respond to T2's request until it reaches a *safe point*: a program point definitely *not* between an Octet barrier and its corresponding program access.

The coordination protocol for conflicting transitions first puts o into an *intermediate* state, which helps simplify the protocol by ensuring that only one thread at a time tries to change an object's state. For example, if T2 wants to read an object that is in the $\mathsf{WrEx_{T1}}$ state, T2 first puts the object into the $\mathsf{RdEx^{Int}_{T2}}$ state. The coordination protocol is then performed in one of two ways:

- The threads perform the *explicit* protocol if T1 is executing code normally. T2 sends a request to T1, and T1 responds to the request when it reaches a safe point. When T2 observes the response, a roundtrip happens-before relationship has been established, so T2 can change the state to $\mathsf{RdEx_{T2}}$ and proceed.

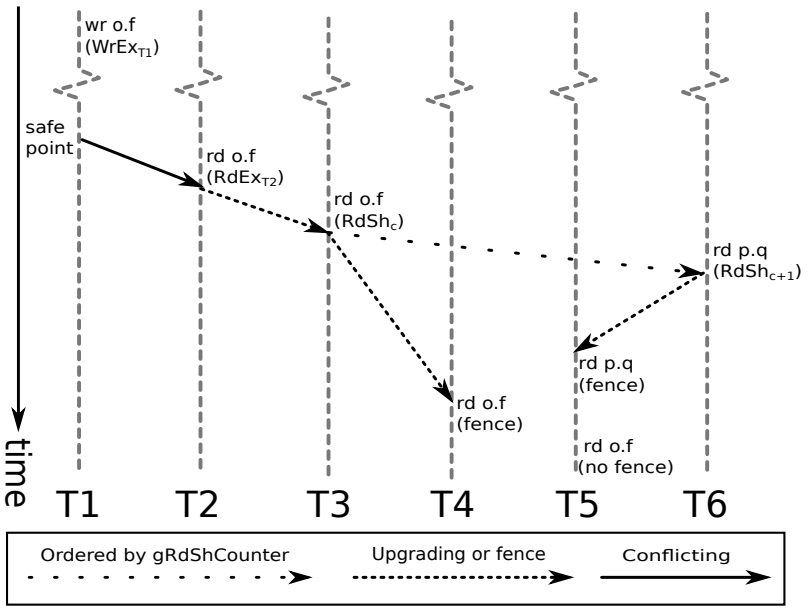


Figure 5.2: A possible interleaving of six concurrent threads accessing shared objects o and p, and the corresponding Octet state transitions they trigger (with new states shown in parentheses).

- Otherwise, thread T1 is "blocking," e.g., waiting for synchronization or I/O. Rather than waiting for T1, T2 *implicitly* coordinates with T1 by atomically setting a flag that T1 will observe when it unblocks. This protocol establishes a happens-before relationship, so T2 can change the state to $\mathsf{RdEx}_{\mathsf{T2}}$ and proceed.

Upgrading and fence transitions. Upgrading and fence transitions (middle rows of Table 5.1) do not require coordination because other threads can safely continue accessing the object under the old state. In Figure 5.2, T3 atomically *upgrades* an object's state from $\mathsf{RdEx}_{\mathsf{T2}}$ to $\mathsf{RdSh}_{\mathsf{c}}$. The value c is the new value of a global counter gRdShCnt that gets incremented atomically every time an object transitions to RdSh, establishing a global order of all transitions to RdSh. This state change establishes a happens-before relationship from

the read on T2 to the current program point on T3, ensuring a transitive happens-before relationship from T1's write to T3's read.

In Figure 5.2, T4 reads o in the RdSh_c state. To ensure a happens-before relationship from the last write to o (by T1) to this read in T4, a *fence* transition is triggered. The fence transition is triggered when a thread's local counter T.rdShCnt is not up-to-date with the counter c in RdSh_c. T4 updates T4.rdShCnt to c and issues a memory fence to ensure a happens-before relationship from T3's transition to RdSh_c to T4's read.

In Figure 5.2, T5 reads o but does *not* trigger a fence transition because T5 has already read an object (p) in the RdSh_{c+1} state. However, a *transitive* happens-before relationship exists from T1's write to T5's read of o because there is a happens-before relationship from o's state transition to RdSh_c in T3 to p's transition to RdSh_{c+1} in T6 (since both transitions update gRdShCnt atomically).

Octet's state transitions thus establish happens-before edges that transitively imply all cross-thread dependences [35]. ICD can piggyback on Octet's state transitions to identify potential cross-thread dependences. Next, we address the challenge of actually identifying the dependence edges that ICD should add to the IDG.

5.3.2 Identifying Cross-Thread Dependences

ICD uses Octet to help detect cross-thread dependences. While Octet establishes happens-before relationships that soundly imply all cross-thread dependences, it does not precisely identify the exact points in the execution with which happens-before relationships are established. ICD addresses the challenge of how to identify these program points and thus add cross-thread edges to the IDG that soundly imply all cross-thread dependences, so that any true dependence cycle will lead to a cycle in the IDG. In this way, ICD detects atomicity

violations soundly but imprecisely with substantially lower overhead than a fully precise approach.

The challenge of identifying each cross-thread edge is in identifying its *source*; the *sink* is obvious since it is the current execution point on the thread triggering the state change. ICD keeps track of a few "last transaction to do X" facts, to help identify sources of cross-thread edges later:

T.lastRdEx – Per-thread variable that stores the last transaction of thread T to transition an object into the $\mathsf{RdEx}_{\mathsf{T}}$ state.

gLastRdSh – Global variable that stores the last transaction among all threads to transition an object into a RdSh state.

We also define the following helper function:

currTX(T) – Returns the transaction currently executing on T.

Creating cross-thread edges for conflicting transitions. A conflicting transition involves one requesting thread reqT, which coordinates with each responding thread respT. ICD piggybacks on each invocation of the coordination protocol, using the procedure handleConflictingTransition() in Figure 5.3, in order to add an edge to the IDG.

Either reqT or respT will invoke the procedure as part of the coordination protocol, depending on whether the explicit or implicit protocol is used. For the explicit protocol, respT invokes the procedure before it responds, which is safe since both threads are stopped at that point. For the implicit protocol, reqT invokes the procedure since respT is blocked; reqT first atomically places a "hold" on respT so respT will not unblock while reqT invokes the procedure.

```
procedure handleConflictingTransition(respT, reqT, oldState, newState)
    IDG.addEdge(currTX(respT) → currTX(reqT))
    if newState = RdEx_reqT then
        reqT.lastRdEx := currTX(reqT)
    end if
end procedure
procedure handleUpgradingTransition(T, oldState, newState)
    Let rdExThread be the thread T such that oldState = RdEx_T
    IDG.addEdge(rdExThread.lastRdEx → currTX(T))
    IDG.addEdge(gLastRdSh → currTX(T))
    gLastRdSh := currTX(T)
end procedure
procedure handleFenceTransition(T)
    IDG.addEdge(gLastRdSh → currTX(T))
end procedure
```

Figure 5.3: ICD procedures called from Octet state transitions.

Figure 5.4 shows a possible thread interleaving among seven concurrent threads executing transactions. The edges among transactions are IDG edges that ICD adds. The access rd o.g in Tx_{2j} conflicts with the first write to object o in transaction Tx_{1i}. The handleConflictingTransition() procedure creates a cross-thread edge in the IDG from Tx_{1i} (the transaction executing the responding safe point) to Tx_{2j} (the transaction triggering the conflicting transition).

To help upgrading transitions (explained next), handleConflictingTransition() updates the per-thread variable T.lastRdEx, the last transaction to put an object into $RdEx_T$. In Figure 5.4, this procedure updates T2.lastRdEx to Tx_{2j} (not shown).

Creating cross-thread edges for upgrading transitions. To see why ICD needs to add cross-thread edges for upgrading transitions (and not just for conflicting transitions), consider

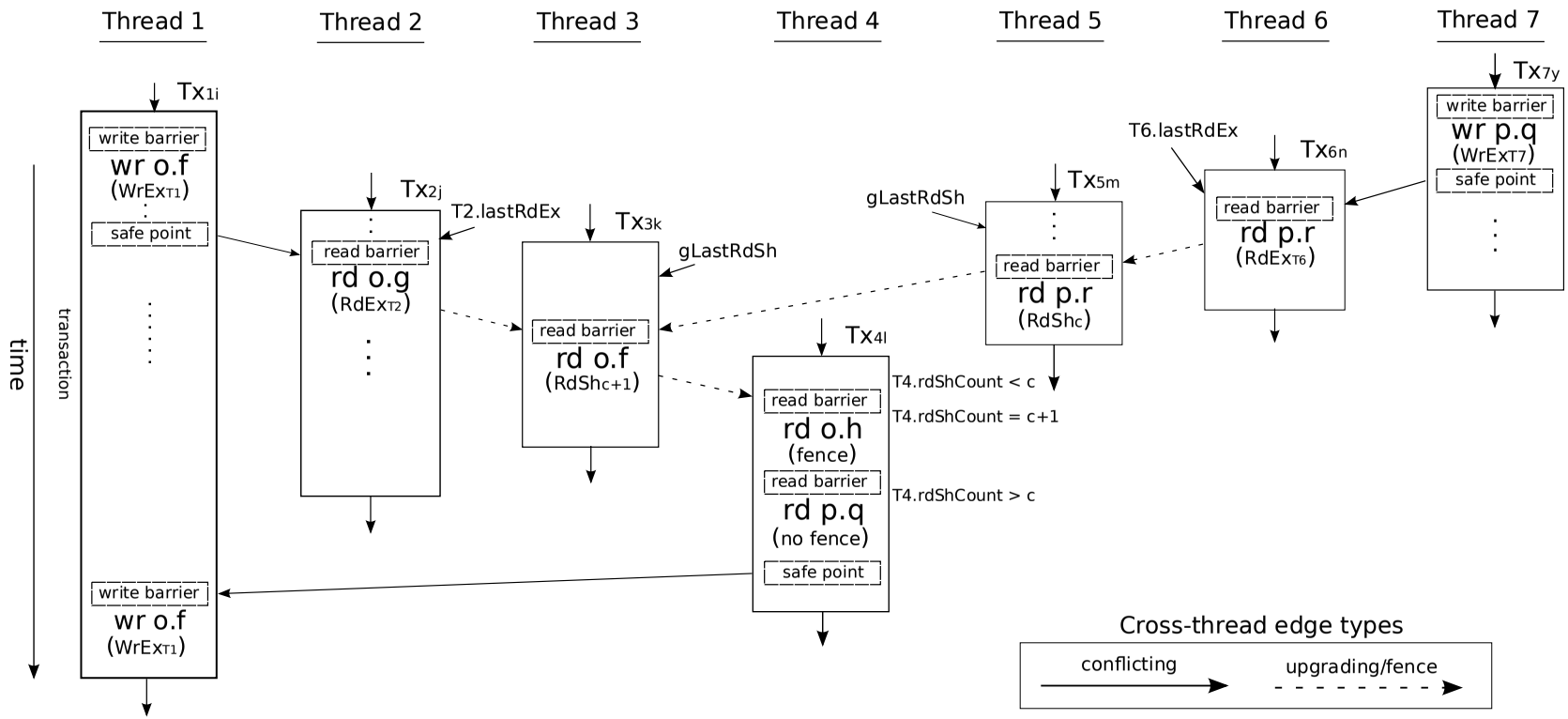


Figure 5.4: An example interleaving of threads executing atomic regions of code as transactions. The figure shows the Octet states after each access and the IDG edges added by ICD.

the upgrading transition from $\mathsf{RdEx}_{\mathsf{T2}}$ to $\mathsf{RdSh}_{\mathsf{c+1}}$ in Figure 5.4. Creating a cross-thread edge is necessary to capture the dependence from T1's write to o to T3's read of o *transitively*. To create this edge, T3 invokes the procedure handleUpgradingTransition() in Figure 5.3.

This procedure also creates a second edge: from the last transaction to transfer an object to the RdSh state, referenced by gLastRdSh, to the current transaction. This edge orders all transitions to RdSh state, and is needed in order to capture dependences related to fence transitions, discussed next. For rd o.f in $\mathsf{Tx}_{\mathsf{3k}}$, the procedure creates an edge from gLastRdSh, which is $\mathsf{Tx}_{\mathsf{5m}}$, to the current transaction. Finally, the procedure updates gLastRdSh to point to the current transaction, $\mathsf{Tx}_{\mathsf{3k}}$.

ICD safely ignores $\mathsf{RdEx}_{\mathsf{T}} \rightarrow \mathsf{WrEx}_{\mathsf{T}}$ upgrading transitions. Any new dependences created by this transition are already captured transitively by existing intra-thread and cross-thread edges.

Creating cross-thread edges for fence transitions. ICD adds edges to the IDG for fence transitions, in order to capture a possible write–read dependence for RdSh objects. Each fence transition calls handleFenceTransition() (Figure 5.3), which creates an edge from the last transaction to transition an object to RdSh (gLastRdSh) to the current transaction.

In Figure 5.4, T4's read of o.h triggers a fence transition and a call to handleFenceTransition(), which creates an edge from gLastRdSh ($\mathsf{Tx}_{\mathsf{3k}}$) to $\mathsf{Tx}_{\mathsf{4l}}$. This edge helps capture the possible dependence from T1's write to T4's read (in this case, no true dependence exists since the accesses are to different fields).

After T4 reads o.h, it reads p.q, which does not trigger a fence transition because T4 has already read an object (o) with a *more recent* RdSh counter (c+1) than p's RdSh counter (c). However, because RdEx $\rightarrow$ RdSh transitions create edges between all transactions that

transition an object to RdSh (e.g., the edge from Tx_{5m} to Tx_{3k}), all write–read dependences are captured by IDG edges even if they do not trigger a fence transition. In the figure, the IDG edges added by the procedures transitively capture the dependence from T7's write to p.q to T4's read of p.q.

Handling synchronization operations. Like Velodrome [73], DoubleChecker captures dependences not only between reads and writes to program variables, but also between synchronization operations: lock release–acquire, notify–wait, and thread fork and join. ICD handles these operations by treating acquire-like operations as reads and release-like operations as writes, on the object being synchronized on.

A distinction with related work is that Farzan and Parthasarathy's analysis (Section 2.3.2) does *not* track synchronization edges [63]. In contrast, DoubleChecker, which follows Velodrome [73], tracks synchronization edges—but tracking synchronization edges can lead to false positives when checking conflict serializability.

Sources of imprecision. ICD is imprecise because the edges it adds to the IDG are imprecise in several ways. First, ICD does not maintain the last transaction to read and write each object, so it identifies last accesses conservatively. For a conflicting transition, ICD adds an edge from the responding thread's last safe point. For an upgrading transition from $\mathsf{RdEx}_{\mathsf{T}}$ to RdSh, it adds an edge from T's last transition to $\mathsf{RdEx}_{\mathsf{T}}$, which may involve a different object.

ICD not only does not maintain the last reader transactions, but it does not maintain even the last reader *threads* for a RdSh object. ICD adds edges between all upgrading transitions to RdSh (to help enable sound tracking of write–read dependences for RdSh objects). For

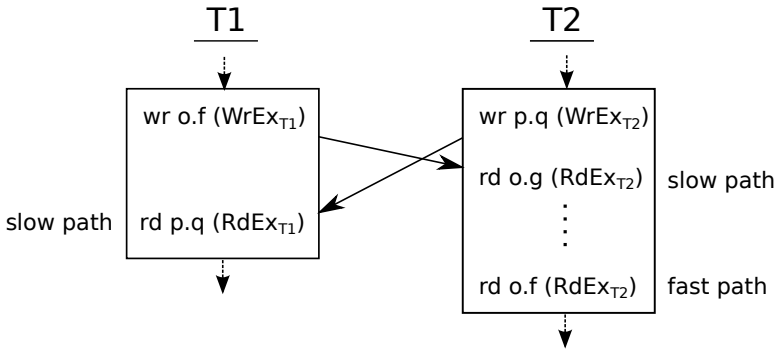
conflicting transitions from RdSh to $\mathsf{WrEx_T}$, ICD adds edges from all threads to T's current transaction.

Finally, ICD tracks dependences at object granularity instead of field granularity.

ICD's imprecision is inherent in its use of Octet, which gives up precise detection of dependences for better performance. Eliminating some but not all sources of ICD's imprecision would be of little use, since ICD would still be imprecise.

5.3.3 Cycle detection

Rather than triggering cycle detection each time it creates a cross-thread edge (as Velodrome does [73]), ICD waits until a transaction ends to detect cycles. Consider the following example.



Even if T1 and T2 each trigger cycle detection when they add cross-thread edges, no precise cycle exists until rd o.f executes. In single-run mode, to ensure that PCD sees the precise cycle, ICD should report the cycle only after the transaction finishes. By invoking cycle detection when transactions end, ICD is guaranteed to detect each cycle at least once. In the first run of multi-run mode, deferring cycle detection until transactions finish is not strictly necessary but leads to fewer invocations of cycle detection.

Detecting strongly connected components. A side effect of delayed cycle detection is that a transaction might be involved in multiple cycles. ICD therefore computes the maximal strongly connected component (SCC) [50] starting from the transaction that just ended, which identifies the set of all transactions that are part of a cycle. The SCC computation explores a transaction tx only if tx has finished. This rule is sound because if tx is indeed involved in cycles, an SCC computation launched when tx finishes will detect those cycles. Avoiding processing unfinished transactions helps prevent identifying the same cycles multiple times, and it avoids races with threads updating their current transaction's state.

In Figure 5.4, ICD detects an SCC (in this case, a simple cycle) of size four when transaction Tx_{1i} ends. In single-run mode or the second run of multi-run mode, ICD passes these transactions to PCD for further processing. Note that PCD detects a precise cycle involving Tx_{1i} and Tx_{3k}. In contrast, if Tx_{3k} did not execute rd o.f, ICD would still detect an imprecise cycle, but PCD would not detect a precise cycle since none exists.

5.3.4 Maintaining Read/Write Logs

In single-run mode or the second run of multi-run mode, when ICD detects a cycle, it passes the set of transactions involved in the cycle to PCD. PCD also needs to know the exact accesses that have executed as well as the cross-thread ordering between them. To provide this information, ICD records read/write logs for every transaction: the exact memory accesses (e.g., object fields) read and written by the transaction. To accomplish this, ICD adds instrumentation before each program access but after Octet's instrumentation that records the access in the current transaction's read/write log. Synchronization operations are recorded as reads or writes to the objects being synchronized on. ICD provides cross-thread ordering of accesses by recording, for each IDG edge, not only the source and sink

transactions of the edge, but also the exact read/write log entries that correspond to the edge's source and sink.

5.3.5 Soundness Argument

We now argue that ICD is a sound first-pass filter. In particular, we show that for any actual (precise) cycle of dependences, there exists an (imprecise) IDG cycle that is a superset of the precise cycle.

Let C be any set of executed nodes $tx_1, tx_2, \ldots, tx_n$ whose (sound and precise) dependence edges form a (sound and precise) cycle $tx_1 \rightarrow tx_2 \rightarrow \ldots \rightarrow tx_n \rightarrow tx_1$.

Let $tx_i \rightarrow tx_j$ be any dependence edge in C. Since ICD adds edges to the IDG that imply all dependences soundly, there must exist a path of edges from tx_i to tx_j in the IDG.

Thus there exists a path $tx_1 \rightarrow tx_2 \rightarrow \ldots \rightarrow tx_n \rightarrow tx_1$ in the IDG. ICD will detect this as a cycle $C' \supseteq C$ and pass C' to PCD. Since C' contains all nodes in C, and PCD computes all dependences between nodes in C', PCD will compute the dependences $tx_1 \rightarrow tx_2 \rightarrow \ldots \rightarrow tx_n \rightarrow tx_1$, and it will thus detect the cycle C.

5.4 Precise Cycle Detection

Precise cycle detection (PCD) is a sound and precise analysis that identifies cycles of dependences on a set of transactions provided as input. DoubleChecker invokes PCD with the following input from ICD: (1) a set of transactions identified by ICD as being involved in an SCC, (2) the read/write logs of the transactions, and (3) the cross-thread edges added by ICD recorded relative to read/write log entries (to order conflicting accesses). PCD processes each SCC identified by ICD separately. Using these inputs, PCD essentially "replays" the subset of execution corresponding to the transactions in the IDG cycle, and performs a sound and precise analysis similar to Velodrome [73]. PCD uses the read/write

```
procedure READ(f, tx)
    if W(f) ≠ null and thread(tx) ≠ thread(W(f)) then
        Add PDG edge: W(f) → tx
    end if
    R(T,f) := tx                                        ▷ Update last read for T
end procedure

procedure WRITE(f, tx)
    if W(f) ≠ null and thread(tx) ≠ thread(W(f)) then
        Add PDG edge: W(f) → tx
    end if
    for all T,  R(T,f) ≠ null do
        if thread(R(T,f)) ≠ thread(tx) then
            Add PDG edge: R(T,f) → tx
        end if
    end for
    W(f) := tx                                          ▷ Update last write
    ∀ T,  R(T,f) := null                                ▷ Clear all reads
end procedure
```

Figure 5.5: Rules to update last-access information for a read and write of field f by a transaction tx.

ordering information to replay accesses in an order that reflects the actual execution order.

As PCD simulates replaying execution from logs, it tracks the last access(es) to each field:

- $\mathcal{W}$(f) is the last transaction to write field f,
- $\mathcal{R}$(T,f) is the last transaction of thread T to read field f.

PCD constructs a *precise dependence graph* (PDG) based on the last-access information. A helper function thread(tx) returns the thread that executes transaction tx. At each read or write of a field f, the analysis (1) adds a cross-thread edge to the PDG (if a dependence exists) and (2) updates the last-access information of f, as shown in Figure 5.5.

PCD detects cycles in the PDG after adding each cross-thread edge. A detected cycle indicates a precise atomicity violation. As part of the error log, PCD reports all the transactions and edges involved in the precise PDG cycle. For example, in Figure 5.4, PCD processes an IDG cycle of size four, computes the PDG, and identifies a precise cycle with just two transactions, Tx_{1i} and Tx_{3k}.

PCD aids debugging by performing *blame assignment* [73], which "blames" a transaction for an atomicity violation if its outgoing edge is created *earlier* than its incoming edge, implying that the transaction *completes* a cycle. In Figure 5.4, PCD blames Tx_{1i}.

5.5 Implementation

We have implemented a prototype of DoubleChecker in Jikes RVM 3.1.3 [9], a high-performance research JVM [20] (Section 3.8.1). Our implementation builds on the publicly available Octet implementation [35].[21] For comparison purposes, we have also implemented Velodrome in Jikes RVM. Flanagan et al.'s implementation [73] is not available, and in any case it is implemented on top of the JVM-independent RoadRunner framework [70], so its performance characteristics could differ significantly. We have made our implementations of DoubleChecker and Velodrome publicly available on the Jikes RVM Research Archive.[21]

5.5.1 DoubleChecker

Specifying atomic regions. DoubleChecker takes an atomicity specification as input, specified as a list of methods to be *excluded* from the specification; all other methods are part of the specification, i.e., they are expected to execute atomically. DoubleChecker extends Jikes RVM's dynamic compilers so each compiled method is statically either *transactional* or *non-transactional*. Methods specified as atomic are always transactional, and non-atomic

[21] `http://www.jikesrvm.org/Research+Archive`

methods are compiled as transactional or non-transactional depending on the caller's context. The compilers compile two versions of non-atomic methods called from both contexts.

Constructing the IDG. The two dynamic compilers in Jikes insert instrumentation to start and end transactions in each *atomic* method called from a *non-transactional* context. At method start, instrumentation creates a new regular transaction. At method end, it creates a new unary transaction. While each non-transactional access conceptually executes in its own unary transaction, our implementation reuses prior work's optimization [73], which merges consecutive unary transactions not interrupted by an incoming or outgoing cross-thread edge.

Each transaction maintains (1) a list of its outgoing edges to other transactions and (2) (for single-run mode or the second run of multi-run mode) a read/write log that is an ordered list of precise memory access entries. Each read/write log entry records information about one access: the base object reference, field address, and read versus write. The read/write log has special entries that correspond to incoming and outgoing cross-thread edges, since PCD needs to know the access order with respect to cross-thread edges.

Transactions and their read/write logs are regular Java objects in our implementation, so garbage collection (GC) naturally collects them as they become transitively unreachable from each thread's current transaction reference. The implementation treats read/write log entries as weak references[22] to avoid memory leaks. When a reference in a read/write log entry dies, our modified GC replaces the reference in the log with the old field address and the current GC invocation count, distinguishing the field precisely.

Our technique, DoubleChecker, and Velodrome do not summarize the dependence graph, but they do rely on garbage collection (GC) to collect transactions not transitively reachable

[22] `http://www.ibm.com/developerworks/java/library/j-jtp11225/`

from any thread's current transaction ("last access" references from objects are treated as weak references), which reduces space overhead in practice.

Still, DoubleChecker can add substantial space overhead, especially to maintain single-run mode's read/write logs. Future work might be able to apply summarization technique [63] to DoubleChecker to reduce space overhead.

Instrumenting program accesses. The compilers add read and write barriers at (object and static) field and array accesses. Our experiments focus on evaluating only instrumenting field accesses and only in application (not Java library) methods, which imitates closely related prior work [67, 73], although we also evaluate the performance of instrumenting array accesses (Section 5.6.4). The compilers instrument program synchronization by treating acquire operations like object reads, and release operations like object writes.

In single-run mode or the second run of multi-run mode, ICD adds instrumentation to record read/write logs. Although logs are ordered, duplicate entries with no incoming or outgoing edges between them can be elided to save space. To elide duplicate entries on the fly, ICD tracks, for each field, the value of a per-thread timestamp of the last access (and whether it was a read or write) to the object; RdSh objects have up to one timestamp per thread. Every time a new transaction starts, or a transaction has an incoming or outgoing edge, a thread increments its current timestamp. It stores this information in per-field metadata for WrEx and RdEx objects and per-thread hash tables for RdSh objects.

5.5.2 Velodrome

Our DoubleChecker and Velodrome implementations share features as much as possible: they instrument the same accesses, demarcate transactions the same way, and represent dependence graphs the same way. The Velodrome implementation does not use Octet. It

adds two words for each object and static field: one references the transaction to write the field, and the other references the last transaction(s) (up to one per thread) to read the field since the last write. To capture release–acquire dependences, each object has an extra header word to track the last transaction to release the object's lock. The implementation treats metadata references as weak references to avoid memory leaks in the transaction dependence graph.

At each access, instrumentation detects cross-thread dependences, adds them to the dependence graph, detects cycles (and reports a precise atomicity violation for each cycle), and updates the field's last-access information. To provide atomicity of the instrumentation together with the program access and thus track cross-thread dependences accurately, the instrumentation and access execute in a small critical section that "locks" a word of the field's metadata using an atomic operation.

5.6 Evaluation

This section evaluates the correctness and performance of our prototype implementation of DoubleChecker in both single- and multi-run modes and compares with Velodrome.

5.6.1 Methodology

Benchmarks. Our experiments run the following programs: the multithreaded DaCapo benchmarks [22] that Jikes RVM 3.1.3 can execute successfully: eclipse6, hsqldb6, lusearch6, xalan6, avrora9, jython9, luindex9, lusearch9,[23] pmd9, sunflow9, and xalan9 (suffixes '6' and '9' distinguish benchmarks from versions 2006-10-MR2 and 9.12-bach, respectively). We also execute the following programs used in prior work [67, 73]: the microbenchmarks

[23] We use a version of lusearch9 that fixes a serious memory leak [191].

elevator, hedc, philo, sor, and tsp [182]; and moldyn, montecarlo, and raytracer from the Java Grande benchmark suite [171].

Experimental setup. We build a high-performance configuration of the JVM (FastAdaptive) that optimizes the JVM, adaptively optimizes the application and uses the default high-performance, generational, stop-the-world Immix garbage collector [23] (GenImmix). We let the JVM adjust the heap size automatically. Our experiments use the *small* workload size of the DaCapo benchmarks, since otherwise DoubleChecker's single-run mode and (to a lesser extent) Velodrome run out of memory with larger workload sizes for a few benchmarks. DoubleChecker's single-run mode also runs out of memory with the standard small size of moldyn and raytracer, so we modify the benchmarks to use an even smaller input, which all atomicity checkers execute. The JVM, which targets the IA-32 platform, is limited to a heap of roughly 1.5–2 GB; a 64-bit implementation could avoid these out-of-memory errors. For the other benchmarks, we use the same input configurations described in prior work [67, 73].

For DoubleChecker's multi-run mode, we execute 10 trials of the first run, take the union of the transactions reported as part of ICD cycles, and use it as input for the second run. This methodology represents a point in the accuracy–performance tradeoff that we anticipate would be used in practice: combining information from multiple first runs should help a second run find more atomicity violations but also increase its overhead.

Platform. The experiments execute on a workstation with a 3.30 GHz 4-core Intel i5 processor, with 4 GB memory running 64-bit RedHat Enterprise Linux 6.5, kernel 2.6.32.

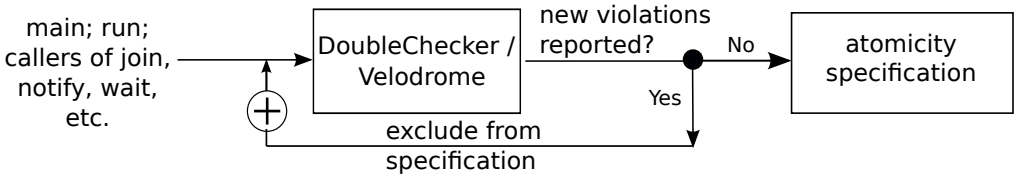


Figure 5.6: Iterative refinement methodology to generate a program's atomicity specification.

Deriving atomicity specifications. Atomicity specifications for the benchmarks either have not been determined by prior work (DaCapo) or are not publicly available (non-DaCapo). We derive specifications for all the programs using an *iterative refinement* methodology used successfully by prior work [67, 72, 73, 184] (Section 2.3.1). Figure 5.6 illustrates iterative refinement. First, all methods are assumed to be atomic with a few exceptions: top-level methods (e.g., main() and Thread.run()) and methods that contain interrupting calls (e.g., to wait() or notify()). We also exclude the DaCapo benchmarks' driver thread (which launches worker threads that actually run the benchmark program) from the atomicity specification, since we know it executes non-atomically. Iterative refinement repeatedly removes methods from the specification when they are "blamed" for detected atomicity violations. We terminate iterative refinement only when no new atomicity violations are reported after 10 trials, in order to approximate well-tested software, which has an accurate atomicity specification and *few, if any, known* atomicity violations.

We use iterative refinement in two ways. First, we use it to evaluate the soundness of DoubleChecker's single- and multi-run modes by comparing the set of atomicity violations reported by Velodrome and DoubleChecker's single- and multi-run modes (Section 5.6.2). For each of the three configurations, we perform iterative refinement to completion and collect all methods blamed as non-atomic along the way.

Second, we use iterative refinement to determine the *final* specifications, i.e., specifications that lead to few or no atomicity violations, in order to evaluate performance (Section 5.6.3). To prepare the final specification for each program, we take the intersection of the finalized specifications (no more violations reported in 10 trials) for both Velodrome and DoubleChecker (single-run mode, since it is fully sound by design), to avoid any bias toward one approach.

We adjust the specifications in a few cases because of out-of-memory errors. raytracer and sunflow9 have one and two long-running transactions, respectively, that execute atomically and that ICD passes to PCD, causing PCD to run out of memory, so we exclude the corresponding methods from the specifications. On the flip side, refining the specification of xalan6 leads to so many transactions being created that DoubleChecker run out of memory, so we use an intermediate (not fully refined) specification for xalan6.

5.6.2 Soundness

DoubleChecker's *single-run* mode is sound and precise by design. By comparing it to Velodrome, we sanity-check our implementations while also observing the effect of timing differences between the two algorithms. *Multi-run* mode is not fully sound, so by comparing it to Velodrome and single-run mode, we evaluate how sound it is in practice. A caveat of this section's comparison is that the first and second runs use the same program inputs, thus representing an upper bound on soundness guarantees.

Table 5.2 shows, for each atomicity checker, the total number of violations reported during all steps of iterative refinement. Each violation in Table 5.2 represents a method identified by blame assignment at least once during this process. At a given step of iterative refinement, a violation reported in one trial may not always be reported in other trials, due

	Velodrome		DoubleChecker		
	Total	**(Unique)**	**Single-run**	**Multi-run**	**(Unique)**
eclipse6	230	(8)	244	190	(8)
hsqldb6	10	(0)	57	57	(0)
lusearch6	1	(0)	1	1	(0)
xalan6	57	(0)	69	54	(0)
avrora9	23	(0)	25	18	(0)
jython9	0	(0)	0	0	(0)
luindex9	0	(0)	0	0	(0)
lusearch9	41	(1)	40	38	(0)
pmd9	0	(0)	0	0	(0)
sunflow9	13	(1)	13	13	(0)
xalan9	78	(0)	82	69	(0)
elevator	2	(0)	2	2	(0)
hedc	3	(1)	3	2	(0)
philo	0	(0)	0	0	(0)
sor	0	(0)	0	0	(0)
tsp	7	(0)	7	7	(0)
moldyn	0	(0)	0	0	(0)
montecarlo	2	(0)	2	2	(0)
raytracer	0	(0)	0	0	(0)
Total	467	(11)	545	453	(8)

Table 5.2: Static atomicity violations reported by our implementations of Velodrome and DoubleChecker. For Velodrome and multi-run mode, *Unique* counts how many violations were *not* reported by single-run mode.

to nondeterministic thread interleavings. Overall, the violations reported by Velodrome and DoubleChecker's single-run mode match closely. Both implementations are sound and precise, so (assuming correct implementations) differences are due to different thread interleavings caused by run-to-run nondeterminism and timing differences between the two analyses. The *Unique* value in parentheses counts violations reported by Velodrome but not by single-run mode; it is nonzero for just four programs, indicating single-run mode finds nearly all violations found by Velodrome. Single-run also mode finds several violations not found by Velodrome. We investigated the program with the greatest discrepancy, hsqldb6.

By inserting random timing delays in Velodrome, we were able to reproduce six violations reported by DoubleChecker, providing some evidence that differences are due to timing effects.

Not surprisingly, multi-run mode does not detect quite as many violations as the sound single-run mode. Overall, multi-run modes detects 83% of all violations detected by single-run mode. Normalizing the detection rate across programs with at least one violation, multi-run mode detects 90% of a program's violations on average, which may be worthwhile in exchange for multi-run mode's lower run-time overhead (discussed next). Multi-run mode finds violations not found by single-run mode only for eclipse6; some but not all of these are the same violations found by Velodrome but not single-run mode.

5.6.3 Performance

This section compares the performance of Velodrome, DoubleChecker's single-run mode, and the first and second runs of DoubleChecker's multi-run mode. We use the final specifications for our performance experiments (Section 5.6.1), and exclude elevator, hedc, and philo, since they are not compute bound [73].

Figure 5.7 shows the execution time of Jikes RVM running various configurations of the Velodrome and DoubleChecker implementations. Execution times are normalized to the first configuration, *Unmodified Jikes RVM*. Each bar is the median of 25 trials to minimize effects of any machine noise. We also show the mean as the center of 95% confidence intervals. Sub-bars show the fraction of time taken by GC.

Velodrome. Our implementation of *Velodrome* slows programs by 6.1X on average. This result corresponds with the 12.7X slowdown reported in prior work [73], although it is hard to compare results since we implement Velodrome in a JVM and use an overlapping

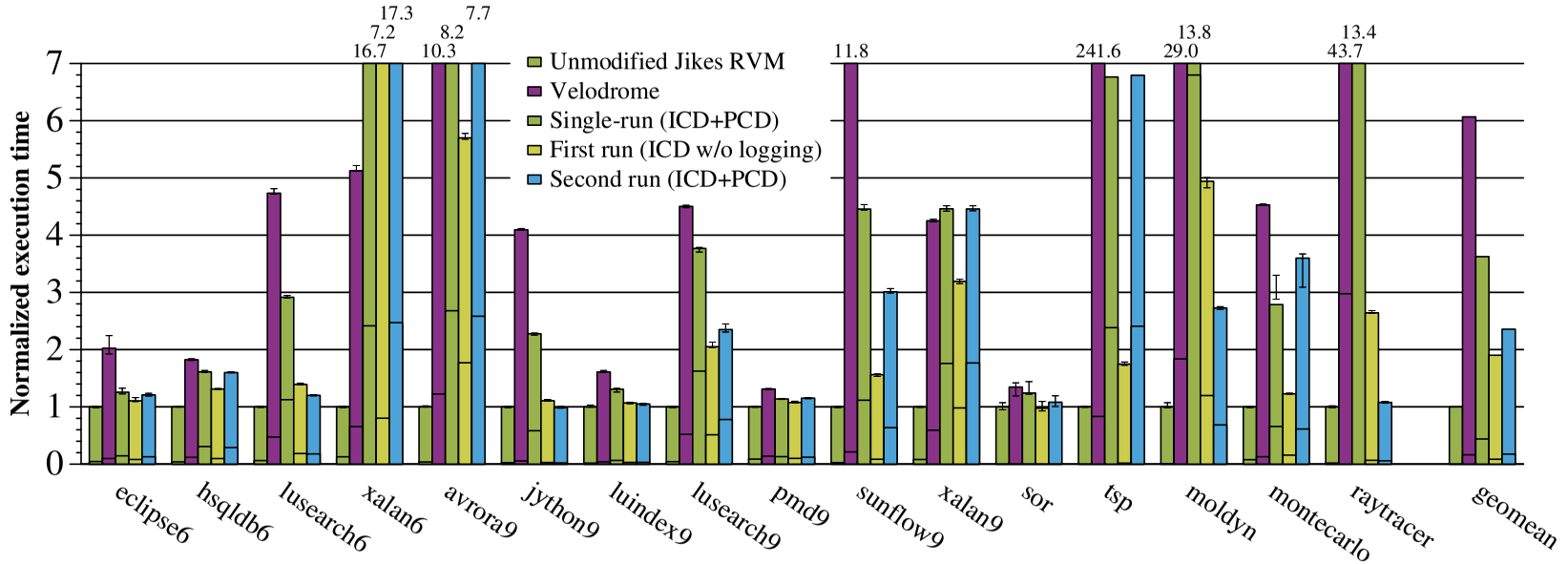


Figure 5.7: Run-time performance comparisons of Velodrome, DoubleChecker's single-run mode, and the first and second runs of DoubleChecker's multi-run mode. The sub-bars show GC time. The geomean GC time excludes short-running sor, which never triggers GC.

but different set of benchmarks. Comparing results for the benchmarks evaluated by prior work, we find that our implementation adds substantially more overhead in several cases. In particular, the Velodrome paper reports 71.7X, 4.5X, and 9.2X slowdowns for tsp, moldyn, and raytracer, respectively [73]. It is somewhat surprising that our implementation in a JVM would add more overhead than the dynamic bytecode instrumentation approach by the Velodrome authors [70, 73]. By running various partial configurations, we find that 82% of this overhead comes from synchronization required to provide analysis–access atomicity, which is unsurprising since atomic operations can lead to remote cache misses on otherwise mostly-read-shared accesses.

According to the Velodrome authors [75], their implementation eschews synchronization when metadata does not actually need to change, i.e., the current transaction is already the last writer or reader. We have implemented and evaluated this variant, which is unsound and can miss dependences in the presence of concurrent accesses, and in fact it crashes on avrora9 due to races accessing metadata. This unsound variant slows programs by 4.1X on average, providing the most help to the programs afflicted most. DoubleChecker still outperforms this unsound variant of Velodrome.

DoubleChecker's single-run mode. The remaining configurations in Figure 5.7 are for DoubleChecker. *Single-run (ICD+PCD)* shows the time incurred to run ICD and PCD in the same execution. This configuration slows programs by 3.6X (260% overhead) on average. Using partial configurations, we find that about two-fifths of this overhead comes from Octet, building the IDG, and detecting IDG cycles. (This partial configuration is similar to the first run of multi-run mode, presented next.) Logging read and write accesses as part of ICD accounts for nearly all of the remaining overhead. Less than one-tenth of the overhead on

average comes from PCD, since ICD filters out most transactions. Single-run mode spends a high fraction of time in GC for several programs, largely because of the memory footprint of long-lived read/write logs. Overall, DoubleChecker's single-run mode avoids much of Velodrome's costs and adds 1.9 times less overhead than Velodrome.

Velodrome outperforms DoubleChecker's single-run mode for one program, xalan6. When executing xalan6, ICD detects many imprecise dependences, triggering SCC detection frequently, and SCC detection finds many imprecise SCCs, leading to high PCD overhead. ICD detects SCCs serially, and PCD detects cycles serially; making them parallel could alleviate this bottleneck.

DoubleChecker's multi-run mode. *First run (ICD w/o logging)* executes only ICD, without logging accesses. Its functionality is similar to a subset of single-run mode evaluated above, and its overhead is unsurprising: it slows programs by 1.9X (90% overhead) on average. The first run of multi-run mode is significantly faster than single-run mode because the former avoids logging.

Second run (ICD+PCD) executes both ICD and PCD (similar to single-run mode), except it only instruments transactions statically identified by the first run, and it instruments non-transactional accesses if and only if the first run identified any non-transactional accesses involved in cycles. The second run slows programs by 2.4X (140% overhead) on average.

We also evaluate a configuration of the second run that always instruments non-transactional accesses—regardless of whether the first run detected any cycles involving unary transactions. Overhead increases to 169%, justifying conditional instrumentation of non-transactional accesses during the second run.

Since the first run detects few imprecise cycles, one might expect the second run would have little work to do. However, the first run identifies transactions *statically* by method signature, leading to many more (dynamic) instrumented accesses in the second run than the total number of (dynamic) accesses identified as involved in cycles in the first run. The filter for non-transactional accesses is even coarser; the second run must instrument all non-transactional accesses in many cases. For this reason, DoubleChecker's ICD and PCD analyses perform better than using Velodrome for the second run, i.e., ICD is still effective as a dynamic transaction filter in the second run. Using Velodrome for the second run (i.e., instrumenting only the transactions statically identified by the first run) slows programs by 2.9X.

A promising direction for future work is to devise an effective way for the first run to more precisely communicate potentially imprecise cycles to the second run.

Summary. Overall, DoubleChecker substantially outperforms prior art. The single-run mode is a fully sound and precise atomicity checker that adds 1.9 times less overhead than Velodrome. Multi-run mode does not guarantee soundness, since atomicity checking is split between two runs, but it avoids the need for logging all program accesses (which the single-run mode needs in order to perform a precise analysis). The first and second runs of the multi-run mode add 5.6 and 3.7 times less overhead than Velodrome, respectively, providing a performance–accuracy tradeoff that is likely worthwhile in practice for providing more acceptable overhead for checking atomicity. DoubleChecker's significant performance benefits justify our design's key insights: it is indeed cheaper to track cross-thread dependences imprecisely in order to filter most of a program's execution from processing by a precise analysis.

5.6.4 Other Performance Investigations

The performance results so far use refined atomicity specifications that lead to no reported atomicity violations. Here we measure the performance of DoubleChecker during iterative refinement. At the beginning of iterative refinement (i.e., the strictest specification), DoubleChecker's single-run mode slows execution by 3.4X. Halfway through iterative refinement (after the first half of the non-atomic methods have been removed from the specification), single-run mode's slowdown is 3.6X. These slowdowns compare closely with the slowdown after full refinement (3.6X), suggesting that performance during iterative refinement is similarly reasonable.

The experiments so far evaluate implementations of DoubleChecker and Velodrome that do not instrument array accesses, since the Velodrome paper also omits this instrumentation [73]. Here we evaluate the additional overhead from array instrumentation. For implementation simplicity, DoubleChecker and Velodrome conflate all elements of an array by using array-level metadata instead of element-level metadata. This makes not only ICD but also Velodrome imprecise, so we disable cycle detection for both analyses. DoubleChecker's single-run mode then runs out of memory for xalan6 and xalan9, so we exclude these programs. DoubleChecker's single-run mode's average slowdown increases from 3.1X (without array instrumentation) to 3.7X (with array instrumentation), and Velodrome's slowdown increases from 6.3X to 7.3X. Note that all four slowdowns skip cycle detection and exclude xalan6 and xalan9.

Finally, to check whether ICD is beneficial as a first-pass filter, we use a "PCD-only" variant of single-run mode in which PCD processes *every* executed transaction, not just transactions identified by ICD's imprecise cycle detection. The PCD-only variant—which is something of a straw man since PCD essentially implements a less-efficient version of

Velodrome's algorithm—increases the slowdown of the single-run mode from 3.1X to 16.6X on average (both results exclude eclipse6, xalan6, avrora9, and xalan9, since the PCD-only variant runs out of memory when running them). This result confirms that ICD is essential as a first-pass filter for PCD.

5.6.5 Run-Time Characteristics

Table 5.3 shows execution characteristics of ICD in single-run mode (the first run of multi-run mode should yield similar results) and the second run of multi-run mode. Each value is the mean of 10 trials of a special statistics-gathering configuration of DoubleChecker; otherwise methodology is the same as Figure 5.7. For each of the two configurations, the table reports the number of regular (non-unary) transactions and accesses instrumented in both regular and unary transactions, and the number of cross-thread edges and SCCs in the IDG. Single-run mode instruments everything, while the second run instruments a subset of transactions. For several programs, the second run avoids instrumenting any accesses because the first run reports no SCCs. For lusearch9 and raytracer, the second run avoids instrumenting any non-transactional accesses since no first-run SCC contained a unary transaction, but non-transactional accesses are instrumented for all the other benchmarks. For programs where the second run instruments (nearly) all transactions and accesses (e.g., xalan6 and avrora9), there is little benefit from multi-run mode's optimization. Even when they should be the same, the counts sometimes differ across modes due to run-to-run nondeterminism.

Compared to how many memory accesses execute, there are few ICD edges, justifying ICD's approach that optimistically assumes accesses are not involved in cross-thread dependences. There are few ICD SCCs in most cases, justifying DoubleChecker's dual-analysis approach and explaining why PCD adds low overhead (except for xalan6; Section 5.6.3).

Name	Single-run mode (or first run of multi-run mode) # Instrumented					Second run of multi-run mode # Instrumented				
	Regular transactions	**Regular accesses**	**Non-trans. accesses**	**# IDG edges**	**# ICD SCCs**	**Regular transactions**	**Regular accesses**	**Non-trans. accesses**	**# IDG edges**	**# ICD SCCs**
eclipse6	793,000	137,000,000	6,610,000	68,400	124	617,000	46,400,000	7,100,000	38,900	80
hsqldb6	87,000	13,400,000	147,000	26,400	76	86,400	10,100,000	148,000	26,200	75
lusearch6	95,700	143,000,000	1,440,000	17	0	0	0	0	0	0
xalan6	1,140,000	70,400,000	17,500,000	211,000	15,500	1,170,000	70,900,000	16,900,000	211,000	15,700
avrora9	22,100,000	264,000,000	362,000,000	2,310,000	854	9,260,000	122,000,000	363,000,000	2,340,000	932
jython9	8	53,200,000	29	0	0	0	0	0	0	0
luindex9	7	8,610,000	25	0	0	0	0	0	0	0
lusearch9	813,000	115,000,000	27,100,000	141	6	64,900	13,500,000	0	142	8
pmd9	7	2,650,000	25	0	0	0	0	0	0	0
sunflow9	35,000	263,000,000	129,000	1,080	25	10,600	176,000,000	129,000	1,020	24
xalan9	1,580,000	67,000,000	14,500,000	66,500	444	1,480,000	66,500,000	15,100,000	67,000	457
elevator	3,200	17,000	5,590	419	24	3,180	16,100	5,590	427	23
hedc	79	38,400	114	83	3	25	37,200	114	85	3
philo	6	16	458	144	0	0	0	0	0	0
sor	2	16	18,700	189	0	0	0	0	0	0
tsp	12,000	386,000	694,000,000	14,100	0	1,340	6,650	691,000,000	11,500	0
moldyn	573,000	194,000,000	50,500,000	38	0	0	0	0	0	0
montecarlo	102,000	179,000,000	93,300,000	30,600	2,860	89,700	145,000,000	108,000,000	30,800	2,730
raytracer	25,700	890,000,000	108,000,000	215	1	4	113	0	9	1

Table 5.3: Run-time characteristics of DoubleChecker for the single-run and the second run in the multi-run mode. Each average is rounded to a whole number with at most three significant digits.

5.7 Contributions and Impact

This work presents a new direction for dynamic sound and precise atomicity checking that divides work into two cooperating analyses: a lightweight analysis that detects cross-thread dependences, and thus atomicity violations, soundly but imprecisely; and a precise analysis that focuses on potential cycles and rules out false positives. These cooperating analyses can execute in a single run, or the imprecise analysis can run alone and inform a second run, providing a performance–soundness tradeoff. The primary contributions of this work are as follows:

- a novel sound and precise dynamic atomicity checker based on using two new, cooperating analyses:
    1. an imprecise analysis that shows it can be cheaper to overapproximate dependence edges rather than compute them precisely, and thus detect cycles whose transactions are a superset of the true (precise) cycles, and
    2. a precise analysis that processes an execution history of only those transactions that are involved in potential cycles;
- two modes of execution that provide two choices for balancing soundness and performance;
- publicly available implementations of DoubleChecker and Velodrome; and
- an evaluation that shows DoubleChecker outperforms current state-of-art, Velodrome, significantly, with multi-run mode providing better performance without sacrificing much soundness in practice.

DoubleChecker outperforms existing sound and precise atomicity checking, suggesting that its new direction has the potential to enable more widespread use of atomicity checking in the real world. For example, an analysis like DoubleChecker can be integrated in a managed language runtime. A programmer can annotate critical regions of code as atomic, and the runtime system can monitor the region for potential atomicity violations. Future IDEs can incorporate support for checking atomicity violations of atomic code regions to flag errors. Such automated support would lead to fewer errors by helping programmers debug and fix concurrency errors arising because of atomicity violations. The end goal is much safer and more reliable concurrent software that would help the society at large.

Chapter 6: Related Work

Chapter 2 covered prior work closely related to data races, memory models, and atomicity. This section presents other work that is related to our proposed techniques, and was not covered in Chapter 2.

Leveraging static analysis. Whole-program static analysis can soundly identify definitely DRF accesses, which dynamic analyses such as data race detectors and dynamic atomicity checkers *need not* instrument. Prior work that takes this approach can reduce the cost of dynamic analysis somewhat but not enough to make it practical for always-on use [49, 57, 106, 182]. These techniques typically use static analyses such as thread escape analysis and thread fork–join analysis. Whole-program static analysis is not well suited for dynamically loaded languages such as Java, since all of the code may not be available in advance.

Our FastTrack, FastRCD, and Valor implementations in Chapter 3 employ intraprocedural static redundancy analysis to identify accesses that do not need instrumentation (Section 3.8.2). These implementations could potentially benefit from more powerful static analyses, although practical considerations (e.g., dynamic class loading and reflection) and inherent high imprecision for large, complex applications, limit the real-world opportunity for using static analysis to optimize dynamic analysis substantially.

Region serializability. Section 2.2.2.2 covered the closest related work on providing SFRSx [20, 115]. Notably, CE and Valor check for conflicting SFRs [20, 115], generating an exception if and only if an execution is not *SFR conflict free* (shown in Figure 4.1). In contrast, RCC provides SFRSx through a combination of conflict checking and consistency enforcement, leading it to allow some executions with SFR conflicts to execute without exceptions (but still guarantee SFR serializability). The figure labels these as the *Exception-free executions in RCC*. RCC is more flexible than CE because it only assumes that access metadata is available at region boundaries and private cache evictions, not eagerly at each access. Unlike CE, Valor and RCC avoids the costs of directly detecting all conflicts by instead inferring some conflicts indirectly (via read validation).

An alternative to detecting region conflicts (i.e., potential violations of region serializability) is to *enforce* end-to-end region serializability. Existing approaches either enforce serializability of full synchronization-free regions (SFRs) [139] or bounded regions [7, 161]. They rely on support for expensive speculation that often requires complex hardware support.

Other dynamic approaches can tolerate the effects of data races by providing isolation from them [149, 151], but the guarantees are limited (Section 2.1).

In other work, Ouyang et al. *enforce* SFR serializability using a speculation-based approach that relies on extra cores to avoid substantial overhead [139].

Other prior work supports memory models based on serializability of *bounded* regions (i.e., weaker than SFRSx) that are in general shorter than full SFRs [7, 42, 116, 121, 161, 167]. These memory models are weaker than SFR serializability, but the corresponding approaches can achieve somewhat lower hardware complexity or run-time overhead, although the hardware still requires extensions to existing cache coherence protocols to support conflict detection and/or speculative execution. To provide end-to-end guarantees, these approaches

also require corresponding compiler modifications to prohibit reordering across region boundaries [7, 121, 167]; or else they provide region serializability for the compiled program only.

Transactional memory. *Transactional memory* (TM) is a general mechanism for providing speculation-based serializable execution of code regions [88, 91]. When TM detects a conflict between regions, instead of generating an exception, it aborts one or more speculatively executing regions (called *transactions*). As such, TM systems need not provide precise conflict detection, whereas SFRSx requires precise conflict detection; on the other hand, to support speculative execution, TM must ensure that for every speculatively written piece of memory, an original (non-speculative) copy exists somewhere in the system. In spite of these differences, TMs fundmantally perform region conflict detection and provide region serializability, and one could imagine adapting TM designs to the problem of SFR serializability, to provide either SFRSx or speculation-based SFR serializability.

Software transactional memory (STM) detects conflicts between programmer-specified regions [86, 88]. To avoid the cost of tracking each variable's last readers, many STMs use so-called "invisible readers" and detect read–write conflicts *lazily* [88]. In particular, *McRT-STM* and *Bartok-STM* detect write–write and write–read conflicts eagerly and read–write conflicts lazily [89, 156]. These STMs validate reads differently from Valor: if a thread detects a version mismatch for an object that it last wrote, it detects a write by an intervening transaction either by looking up the version in a write log [156], or by waiting to update versions until a transaction ends (which requires read validation to check each object's ownership) [89].

In contrast, Valor avoids the costs of maintaining this data by checking if the version has increased by at least two. Another difference is that Valor must detect conflicts precisely, whereas STMs do not (a false conflict triggers an unnecessary abort and retry). As a result, STMs typically track conflicts at the granularity of objects or cache lines. More generally, STMs have not introduced designs that target region conflict detection or precise exceptions. In some sense, our work applies insights from STMs to the context of data race exceptions.

Some software transactional memory (STM) systems have used mechanisms related to RCC's mechanisms, such as version and value validation of reads [52, 87, 89, 138, 156]. Like RCC, *NOrec* and *JudoSTM* use value-based validation [52, 138]. NOrec buffers writes and validates read values lazily, which is similar in spirit to RCC's mechanism for private cache lines [52]. NOrec uses a global sequence lock to ensure consistency of read validation and writing back of buffered writes. RCC avoids a global sequence lock by (1) dividing performing writes into pre-commit and post-commit and (2) using versions to achieve atomic read validation (solving the ABA problem[24]). Related to RCC's read validation scheme, Harris and Fraser use value validation as a "second chance" for a committing transaction that fails version validation [87]. The purpose is to avoid aborts and allow more concurrency, whereas RCC's read validation scheme is required in order to avoid false aborts.

Hammond et al. introduce a memory model and associated hardware called *Transactional memory Coherence and Consistency* (TCC) that executes *all* code in transactions [85]. The programmer specifies TCC transactions and hardware speculatively executes them using conflict detection and re-execution to enforce atomicity. Speculative execution allows TCC to track access information for memory regions coarser than a byte (e.g., cache line), but coarse meta-data tracking puts TCC at risk for false conflicts.

[24] `https://en.wikipedia.org/wiki/ABA_problem`

Like RCC, TCC does not use a conventional cache coherence protocol to detect access conflicts; instead it uses an update-broadcast system [88]. In update-broadcast systems, if a transaction reaches its commit point successfully, then it notifies other processors of its write set. Conversely, if a processor receives a write-set notification which conflicts with its own read-set, then the processor aborts its own transaction. TCC broadcasts a transaction's write set at the end of the transaction to detect conflicts. Follow-up work to TCC shows that this operation that can be made less inefficient by using a directory [44] and introducing parallel commit [148]. Despite these optimizations, TCC is fundamentally limited by its use of bounded eviction and write buffers. When a transaction overflows a buffer, the executing core must execute non-speculatively, with exclusive commit access from the point of overflow to the end of the transaction [88]. Non-speculative, exclusive execution impedes parallelism and performance, as we showed in Section 4.6.

RCC's novel algorithm, mechanism, and architecture address the essential flaws in TCC. Unlike TCC, RCC transactions are defined by existing synchronization operations and RCC tracks precise, byte-granular access information. RCC's novel algorithm and mechanism avoids substantial conflict detection cost via read validation (Section 4.3). RCC avoids speculation, as well as the buffer bounding limitations in its implementation, eliminating a key impediment to parallel performance.

Like TCC, *BulkSC* executes bounded regions transactionally, using conflict detection to enforce coherence and consistency [42, 43]. BulkSC's regions are dynamically formed and bounded, and its conflict detection mechanism works by brodcasting a completing region's write set, not relying on existing cache coherence support. To ensure progress, BulkSC dynamically resizes regions, making it inadequate to support SFRSx, which requires fixed regions.

In contrast with BulkSC, with TCC and its variants, and with RCC, most hardware TMs (HTMs) build on M(O)ESI-based cache coherence protocols to detect and resolve conflicts [10, 26, 27, 128, 192]. The key challenge encountered by these designs, if they support unbounded transactions, is when a transaction's working set overflows the private cache(s): a conflict on non-privately-cached data does not generate coherence events. These HTM designs incur substantial cost and complexity to maintain state for overflowed bits and detect conflicts as they occur. For example, *LogTM* extends the directory with "sticky" coherence states for lines in order to detect conflicts on lines that overflow a transaction [128]. In contrast, RCC allows regions to execute largely independently, detecting and inferring conflicts lazily (upon a cache miss or region boundary).

Other HTMs support conflict detection and speculative execution only for *bounded* transactions [91, 193], relying on a software TM (STM) fallback [12, 38, 102, 127]. Intel's recent processors provide bounded, best-effort hardware TM; implementations do not support transactions that overflow private caches [193]. *RaceTM* uses *hardware* TM to detect conflicts that are data races [83]. RaceTM is thus closest to existing hardware-based conflict detection mechanisms [115, 121, 167].

Detecting conflicts. *Last writer slicing* (LWS) tracks data provenance, recording only the last writers of data [114]. LWS and our work share the intuition that to achieve low run-time overheads, a dynamic analysis should track only the last writer of each shared memory location. LWS is considerably different from our work in its purpose, focusing on understanding concurrency bugs by directly exposing last writer information in a debugger. LWS cannot detect read–write conflicts, and it does not detect races or provide execution model guarantees.

Rethinking cache coherence. Pervasive cache coherence protocols propagate memory access information across cores eagerly in order to provide hardware consistency models. This functionality requires eagerly invalidating shared lines in remote caches and also requires the directory to maintain a list of read sharers. Recently, there have been efforts to reduce the complexity of coherence protocols and the memory subsystem design by relying on disciplined parallelism [48, 100]. *SARC* tries to reduce invalidation traffic on writes by creating *tear-off* copies of shared lines [100]. SARC reduces directory indirection by trying to predict writers. The SARC protocol still needs to communicate eagerly for cache lines that are not sent to sharers as tear-off copies, and thus complicates the existing protocol.

DeNovo shows that a simple coherence protocol can provide consistency for data-race-free (DRF) programs [48]. In addition to requiring DRF executions, DeNovo requires explicit programmer annotations to identify memory regions that need to be self-invalidated at synchronization boundaries. In contrast, RCC provides consistency for both DRF and racy executions, and it requires no annotations. DeNovo's requirement of DRF restricts the synchronization constructs that can be used in programs, since invalidations in DeNovo are initiated by readers. This constraint is relaxed in follow-up work [176, 177]. *DeNovoND* can allow for well-structured locks with custom hardware for such locks [177], while *DeNovoSync* adds single-reader constraints to serialize reads and writes [176].

TSO-CC is a coherence protocol geared toward providing TSO that does not track sharers, and relies on self-invalidation and detection of potential acquire operations [58]. The *VIPS* protocol attempts to simplify MESI by distinguishing shared from private data [154]. For private data, VIPS uses a write-back policy, while shared data is write-through. Furthermore, only shared data is flushed from private caches at synchronization points. However, the work provides coherence for DRF executions only.

Distributed shared memory. Some distributed shared memory (DSM) systems provide *release consistency*, which allows deferring coherence operations until synchronization operations [3, 15, 25, 40, 64, 101]. While RCC's mechanisms are inspired by release consistency mechanisms, RCC's design is substantially different in order to provide strong guarantees for racy executions, whereas release consistency assumes data race freedom. Release consistency mechanisms thus do not require nor include features of RCC's such as tracking of access metadata and versions, and read validation and pre-commit operations. DSM systems are thus intellectually closer to recent work that assumes data race freedom, such as DeNovo and SARC [48, 100], than to RCC.

Chapter 7: Future Work

According to Griffiths (page 97) [82], "Perfection in research is not be found. . . . There is no hope of doing perfect research." Griffiths refused the presence of perfect research in any field of study. According to him, as and when new techniques and innovations are made in a field of study, the old research becomes outdated and ends up being subsumed by the newer techniques. I agree with Griffiths' observation: the techniques presented in this dissertation are not perfect either. We are aware of a few opportunities for improvement in each of the three techniques presented, and there are possibly several more that readers might think of. In the following, we have discussed possible improvements and extensions to each technique.

7.1 Valor

In this dissertation, we have advocated for and presented the advantages of exception-based memory model semantics (Section 2.2.2). For example, SFRSx provides well-defined semantics even for racy executions. Under SFRSx, data races are potential fail-stop errors, which is analogous to fixing existing memory errors, such as buffer overflows and null pointer exceptions, for which memory- and type-safe languages such as Java provide fail-stop semantics. 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). Nonetheless, even well-tested software may contain unknown data races that may manifest only under certain production environments, inputs, or thread interleavings [107, 145, 179] (Section 2.1), leading to unexpected consistency exceptions which may frustrate developers and users more than today's often silent and unknown consequences under DRF0-based memory models. Any racy execution can throw a consistency exception, essentially trading *availability* for strong, well-defined semantics. Unexpected exceptions hurt the availability (i.e., exception freedom) of production systems.

Future research could target to extend FastRCD and Valor or develop new analysis to check if consistency exceptions can be tolerated, thereby improving availability [195], and still provide strong memory consistency based on a some notion of region isolation.

7.2 RCC

A requirement for providing the SFRSx memory model in hardware is to precisely track access metadata at the granularity of bytes. The RCC design achieves this by extending the private cache lines to store access metadata. Since it is impractical to store the access information for all cores for each LLC line in the LLC, therefore, RCC adds a metadata cache, called the AIM cache, which is connected on a bus shared with the LLC. However, the AIM cache might not scale well with *large* number of cores, since a larger AIM structure would imply increased area-on-chip and energy consumption. Furthermore, large region sizes and large number of cores might lead to more AIM cache misses with reasonable-sized AIM caches, which is expensive (e.g., canneal in Section 4.6.2).

A potential direction of future work is to provide alternative memory models that are stronger than DRF0-based variants and where the access information can be approximated (e.g., tracked imprecisely). This would mean that the AIM cache is no longer required,

implying fewer customizations to current commodity processors. That would also reduce the required area-on-chip and the energy consumption in the new design.

7.3 DoubleChecker

In the following, we present possible extensions to DoubleChecker:

- A conflict serializability-based atomicity checker needs to check for presence of cycles in the transactional dependence graph [18, 73]. DoubleChecker performs lazy cycle detection, and hence checks for presence of strongly connected components (SCCs) in the dependence graph (Section 5.3.3). Our evaluation shows that there is some scope to improve the overhead of detecting SCCs during imprecise cycle detection phase. Our current implementation of cycle detection is single-threaded, for implementation simplicity. A possible extension could be to investigate concurrent cycle detection algorithms from the reference counting-based garbage collection literature [144], and improve the implementation.

- A potential drawback of DoubleChecker is that it needs to log program accesses to pass from the imprecise to the precise analysis, in order to recover precision. This stresses the memory requirements of the approach, and increases garbage collection overhead. A potential area of work could explore summarization techniques that would analyze the transactions on the fly and dynamically shrink the read/write logs by soundly dropping accesses that cannot be part of a future atomicity violation [63].

Chapter 8: Conclusion

Current shared-memory systems are afflicted by a critical problem, namely poor and unsatisfactory semantics for racy program executions. Given that there are no known techniques to deal with all data races satisfactorily, it is imperative to strengthen existing memory models to provide strong semantic guarantees to all program executions.

In this dissertation, we propose efficient techniques to provide strong region serializability-based memory consistency. This dissertation proposes two solutions to solve the problem by providing a strong memory model called SFRSx to all program executions, by efficiently detecting and/or inferring conflicting accesses between SFRs. In addition, we extend the scope of the dissertation's work by providing serializability of programmer-defined atomic regions.

In the following, we summarize this dissertation's contributions and discuss how the techniques presented help in solving the stated problem.

8.1 Summary

- **Chapter 3** presents two new software-based region conflict detectors, one of which, Valor, has overheads low enough to provide practical semantic guarantees to a language specification. The key insight behind Valor is that detecting read–write conflicts lazily retains necessary semantic guarantees and has better performance than eager

conflict detection. Valor provides the SFRSx memory model end-to-end in software, and its overhead is potentially low enough to use all-the-time conflict exceptions in various settings, from in-house testing to alpha and beta testing to potentially even some production systems. Overall, Valor represents an advance in the state of the art for providing strong semantic guarantees for racy executions.

- **Chapter 4** presents RCC, an architecture design that provides end-to-end SFRSx, ensuring strong, well-defined semantics for all executions, including executions with data races. RCC's insights and contributions lie in its novel mechanism for detecting regions conflicts that ensures write atomicity and checks read consistency. RCC's support for region serializability enables coherence to be deferred to SFR boundaries. An unoptimized design of RCC incurs significant costs to provide coherence conservatively; we show how optimizations reduce these costs substantially.

  Our evaluation shows that RCC compares favorably, in terms of execution time and on- and off-chip traffic, with prior work that detects region conflicts for memory consistency or transactional memory; RCC's novel mechanism avoids the high costs and scalability bottlenecks incurred by the prior work's mechanisms [85, 115]. Furthermore, RCC is competitive with the evaluation baseline that provides no conflict detection support or strong consistency guarantees. By eschewing the limitations that make prior work unimplementable, and providing significantly stronger guarantees than current shared-memory systems at comparable cost, RCC is a compelling design for future systems and advances the state of the art in parallel architecture consistency and coherence.

- **Chapter 5** presents DoubleChecker, an efficient sound and precise dynamic software-only atomicity checking technique. DoubleChecker is a novel technique that presents a new direction for dynamic sound and precise atomicity checking to provide serializability of arbitrarily-sized regions. Specifically, we show that it is often cheaper to imprecisely track cross-thread dependences and speed up the common case of no dependences between accesses. The key insight of DoubleChecker lies in its separation of concerns into two new staged and cooperating dynamic analyses—an imprecise and a precise analysis. Its imprecise analysis tracks cross-thread dependences soundly but imprecisely with significantly better performance than a fully precise analysis. Its precise analysis is more expensive but only needs to process a subset of the execution identified as potentially involved in atomicity violations by the imprecise analysis. This separation of concerns allows DoubleChecker to improve the performance of dynamic atomicity checking substantially compared with existing state-of-the-art detectors (e.g., [73]).

8.2 Impact and Meaning

Software is pervasive in our modern society, with an aim to aid human beings in every aspect of their daily lives. Software is everywhere, ranging from complex machines like nuclear reactors, space shuttles, cars, to the more simple devices and household appliances such as microwaves, wrist watches, etc. The ultimate goal is for software to be a boon to humanity and not a bane. A prerequisite for software to be a boon is "software that does what it is intended for," i.e., software of *good quality*. Unfortunately, as Mark Paulk from Carnegie Mellon University's Software Engineering Institute noted, "A fundamental

problem with software quality is that programmers make mistakes."[25] Programming is a human activity and humans are prone to making errors [92].

We are beyond a "point of no return" with the prevalence of parallel computer systems. To serve critically important health, resource discovery, economic, and security applications with the extreme scale and performance afforded by today's parallel hardware, developers are faced with the daunting, complex task of writing parallel software. Today's inscrutable and inadequate programming language memory consistency models lead to errors that, in the absence of (intractable) exhaustive testing, persist in deployed parallel software. System failures stemming from *data races* have real financial [145] and social [107, 179] costs, and finding and fixing software errors is the dominant cost of a computer system (Chapter 2).

Memory models for current languages have largely been designed from a system-centric point of view that complies to optimizations originally designed for sequential languages [119]. According to Adve and Boehm [2],

> *[I]t is time to rethink how we design our languages and systems.*
>
> ...
>
> *We also believe some of the messiness of memory models today could have been averted with closer cooperation between hardware and software.*

Our work certainly fits the theme championed by prior research [2, 41, 115]. The techniques proposed in this proposal have the potential to be the foundation on which future languages and systems are based to provide SFR serializability to all program executions. As discussed, such strong guarantees help simplify programming language specifications, permits aggressive optimizations by the compiler and the hardware within SFRs, and they help debug problematic data races by making them a fail-stop condition. The work in

[25] Information Week, Issue on Software Quality, January 21 2002.

this dissertation will help benefit society by fundamentally simplifying programming and increasing reliability of software for parallel computers. The techniques will, if adopted successfully, address a long-standing problem for parallel computing systems, leading to more reliable and scalable systems, benefiting all aspects of society that rely on computation.

Bibliography

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *TOPLAS*, 28(2):207–255, 2006.

[2] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.

[3] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *HPCA*, pages 26–37, 1996.

[4] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1996.

[5] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *ISCA*, pages 2–14, 1990.

[6] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *ISCA*, pages 234–243, 1991.

[7] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. BulkCompiler: High-performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *MICRO*, pages 133–144, 2009.

[8] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In *DAMP*, pages 13–24, 2008.

[9] 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.

[10] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA*, pages 316–327, 2005.

[11] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-Run Software Failure Diagnosis via Hardware Performance Counters. In *ASPLOS*, pages 101–112, 2013.

[12] L. Baugh, N. Neelakantam, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *ISCA*, pages 115–126, 2008.

[13] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, pages 53–64, 2010.

[14] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, pages 81–96, 2009.

[15] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, 1991.

[16] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, pages 72–81, 2008.

[18] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *PLDI*, pages 28–39, 2014.

[19] S. Biswas, M. Zhang, and M. D. Bond. Lightweight Data Race Detection for Production Runs. Technical Report OSU-CISRC-1/15-TR01, Computer Science & Engineering, Ohio State University, 2015. `ftp://ftp.cse.ohio-state.edu/pub/tech-report/2015/TR01.pdf`.

[20] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.

[21] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE*, pages 137–146, 2004.

[22] 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 *OOPSLA*, pages 169–190, 2006.

[23] S. M. Blackburn and K. S. McKinley. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *PLDI*, pages 22–32, 2008.

[24] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13:422–426, 1970.

[25] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *ISCA*, pages 142–153, 1994.

[26] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *ISCA*, pages 24–34, New York, NY, USA, 2007.

[27] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *ISCA*, pages 127–138, 2008.

[28] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must Be Deterministic by Default. In *HotPar*, pages 4–9, 2009.

[29] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, 2011.

[30] H.-J. Boehm. Position paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *RACES*, pages 9–14, 2012.

[31] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.

[32] H.-J. Boehm and S. V. Adve. You Don't Know Jack about Shared Variables or Memory Models. *CACM*, 55(2):48–54, 2012.

[33] H.-J. Boehm and B. Demsky. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *MSPC*, pages 7:1–7:6, 2014.

[34] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional Detection of Data Races. In *PLDI*, pages 255–268, 2010.

[35] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *OOPSLA*, pages 693–712, 2013.

[36] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, pages 211–230, 2002.

[37] J. Burnim, K. Sen, and C. Stergiou. Testing Concurrent Programs on Relaxed Memory Models. In *ISSTA*, pages 122–132, 2011.

[38] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *PACT*, pages 187–200, 2014.

[39] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *CACM*, 51(11):40–46, 2008.

[40] M. Castro, P. Guedes, M. Sequeira, and M. Costa. Efficient and Flexible Object Sharing. In *ICPP*, August 1996.

[41] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. A Case for System Support for Concurrency Exceptions. In *HotPar*, 2009.

[42] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, pages 278–289, 2007.

[43] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, pages 227–238, 2006.

[44] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *HPCA*, pages 97–108, 2007.

[45] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In *FASE*, pages 425–439, 2009.

[46] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring Locks for Atomic Sections. In *PLDI*, pages 304–315, 2008.

[47] L. Chew and D. Lie. Kivati: Fast Detection and Prevention of Atomicity Violations. In *EuroSys*, pages 307–320, 2010.

[48] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT*, pages 155–166, 2011.

[49] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, pages 258–269, 2002.

[50] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 11. The MIT Press, McGraw-Hill Book Company, 2nd edition, 2001.

[51] L. Dalessandro and M. L. Scott. Sandboxing Transactional Memory. In *PACT*, pages 171–180, 2012.

[52] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, pages 67–78, 2010.

[53] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, pages 85–96, 2009.

[54] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *ISCA*, pages 201–212, 2012.

[55] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *OOPSLA*, pages 355–372, 2013.

[56] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *OOPSLA*, pages 467–484, 2012.

[57] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, pages 245–255, 2007.

[58] M. Elver and V. Nagarajan. TSO-CC: Consistency directed cache coherence for TSO. In *HPCA*, pages 165–176, 2014.

[59] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, pages 237–252, 2003.

[60] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, pages 1–16, 2010.

[61] M. Eslamimehr and J. Palsberg. Race Directed Scheduling of Concurrent Programs. In *PPoPP*, pages 301–314, 2014.

[62] A. Farzan and P. Madhusudan. Causal Atomicity. In *CAV*, pages 315–328, 2006.

[63] A. Farzan and P. Madhusudan. Monitoring Atomicity in Concurrent Programs. In *CAV*, pages 52–65, 2008.

[64] C. Fensch and M. Cintra. An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs. In *HPCA*, pages 355–366, 2008.

[65] C. Flanagan. Verifying Commit-Atomicity Using Model-Checking. In *SPIN*, pages 252–266, 2004.

[66] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *PLDI*, pages 219–232, 2000.

[67] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. *SCP*, 71(2):89–109, 2008.

[68] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[69] C. Flanagan and S. N. Freund. Adversarial Memory for Detecting Destructive Races. In *PLDI*, pages 244–254, 2010.

[70] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*, pages 1–8, 2010.

[71] C. Flanagan and S. N. Freund. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *ECOOP*, pages 255–280, 2013.

[72] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for Atomicity: Static Checking and Inference for Java. *TOPLAS*, 30(4):20:1–20:53, 2008.

[73] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, pages 293–303, 2008.

[74] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *PLDI*, pages 338–349, 2003.

[75] S. Freund, 2013. Personal communication.

[76] S. N. Freund, 2015. Personal communication.

[77] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *SPAA*, pages 316–326, 1991.

[78] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-memory Multiprocessors. In *ASPLOS*, pages 245–257, 1991.

[79] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *ISCA*, pages 15–26, 1990.

[80] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An Exploratory Survey. In EC^2, 2008.

[81] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-Driven Software Race Detection using Hardware Performance Counters. In *ISCA*, pages 165–176, 2011.

[82] M. Griffiths. *Educational Research For Social Justice: getting off the fence*. Open University Press, 1998.

[83] S. Gupta, F. Sultan, S. Cadambi, F. Ivančić, and M. Rötteler. Using Hardware Transactional Memory for Data Race Detection. In *IPDPS*, pages 1–11, 2009.

[84] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic Detection of Atomic-Set-Serializability Violations. In *ICSE*, pages 231–240, 2008.

[85] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, pages 102–113, 2004.

[86] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, pages 388–402, 2003.

[87] T. Harris and K. Fraser. Revocable Locks for Non-Blocking Programming. In *PPoPP*, pages 72–82, 2005.

[88] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[89] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *PLDI*, pages 14–25, 2006.

[90] J. Hatcliff, Robby, and M. B. Dwyer. Verifying Atomicity Specifications for Concurrent Object-Oriented Software using Model-Checking. In *VMCAI*, pages 175–190, 2004.

[91] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[92] G. J. Holzmann. The Logic of Bugs. In *FSE*, pages 81–87, 2002.

[93] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two Hardware-Based Approaches for Deterministic Multiprocessor Replay. *CACM*, 52:93–100, 2009.

[94] J. Huang, P. O. Meredith, and G. Rosu. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *PLDI*, pages 337–348, 2014.

[95] Intel. Intel® Core™ i7-3970X Processor Extreme Edition. `http://ark.intel.com/products/70845`.

[96] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated Atomicity-Violation Fixing. In *PLDI*, pages 389–400, 2011.

[97] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *OSDI*, pages 295–308, 2008.

[98] B. Kasikci, C. Zamfir, and G. Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, pages 185–198, 2012.

[99] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced Data Race Detection. In *SOSP*, pages 406–422, 2013.

[100] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54–65, 2010.

[101] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, pages 13–21, 1992.

[102] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *PPoPP*, pages 209–220, 2006.

[103] H. Labs. CACTI 5.3. `http://quid.hpl.hp.com:9081/cacti/`.

[104] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

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

[106] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *PLDI*, pages 463–474, 2012.

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

[108] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California at Berkeley, 2004.

[109] C. Lin, V. Nagarajan, and R. Gupta. Efficient Sequential Consistency Using Conditional Fences. In *PACT*, pages 295–306, 2010.

[110] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS*, pages 273–286, 2012.

[111] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 2nd edition, 1999.

[112] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[113] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ASPLOS*, pages 37–48, 2006.

[114] B. Lucia and L. Ceze. Data Provenance Tracking for Concurrent Programs. In *CGO*, pages 146–156, 2015.

[115] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[116] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, pages 277–288, 2008.

[117] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, pages 190–200, 2005.

[118] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[119] D. Marino, T. D. Millstein, M. Musuvathi, S. Narayanasamy, and A. Singh. The Silently Shifting Semicolon. In *SNAPL*, pages 177–189, May 2015.

[120] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, pages 134–143, 2009.

[121] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[122] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-Preserving Compiler. In *PLDI*, pages 199–210, 2011.

[123] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-Chip Cache Coherence is Here to Stay. *CACM*, 55(7):78–89, July 2012.

[124] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *MICRO*, pages 3–14, 2002.

[125] H. S. Matar, I. Kuru, S. Tasiran, and R. Dementiev. Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support. In *WoDet*, 2014.

[126] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1988.

[127] A. Matveev and N. Shavit. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *ASPLOS*, pages 59–71, 2015.

[128] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265, 2006.

[129] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.

[130] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, pages 337–348, 2009.

[131] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *POPL*, pages 327–338, 2007.

[132] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI*, pages 308–319, 2006.

[133] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*, pages 22–31, 2007.

[134] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, pages 89–100, 2007.

[135] H. Nishiyama. Detecting Data Races using Dynamic Escape Analysis based on Read Barrier. In *VMRT*, pages 127–138, 2004.

[136] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *PPoPP*, pages 167–178, 2003.

[137] C. O'Hanlon. A Conversation with John Hennessy and David Patterson. *Queue*, 4(10):14–22, Dec. 2006.

[138] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *PACT*, pages 365–375, 2007.

[139] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. ...and region serializability for all. In *HotPar*, 2013.

[140] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, 1979.

[141] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ISCA*, pages 348–354, 1984.

[142] C.-S. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *FSE*, pages 135–145, 2008.

[143] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS*, pages 25–36, 2009.

[144] H. Paz, D. F. Bacon, E. K. Kolodner, E. Petrank, and V. T. Rajan. An Efficient On-the-Fly Cycle Collection. *TOPLAS*, 29(4):1–43, 2007.

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

[146] E. Pozniansky and A. Schuster. MultiRace: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *CCPE*, 19(3):327–340, 2007.

[147] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, pages 320–331, 2006.

[148] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and Reliable Communication for Hardware Transactional Memory. In *PACT*, pages 144–154, 2008.

[149] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs. In *ASPLOS*, pages 181–192, 2009.

[150] P. Ranganathan, V. Pai, and S. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *SPAA*, pages 199–210, 1997.

[151] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and Tolerating Asymmetric Races. In *PPoPP*, pages 173–184, 2009.

[152] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *TOPLAS*, 20:483–545, 1998.

[153] C. G. Ritson and F. R. Barnes. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *CPA*, pages 271–292, 2013.

[154] A. Ros and S. Kaxiras. Complexity-Effective Multicore Coherence. In *PACT*, pages 241–252, 2012.

[155] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming Device Drivers. In *EuroSys*, pages 275–288, 2009.

[156] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.

[157] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER Multiprocessors. In *PLDI*, pages 175–186, 2011.

[158] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, pages 27–37, 1997.

[159] C. Segulja and T. S. Abdelrahman. Clean: A Race Detector with Cleaner Semantics. In *ISCA*, pages 401–413, 2015.

[160] K. Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, pages 11–21, 2008.

[161] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, pages 561–575, 2015.

[162] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic Race Detection with LLVM Compiler. pages 110–114, 2012.

[163] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, pages 27–51, 2008.

[164] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *CACM*, 53(7):89–97, 2010.

[165] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *TOPLAS*, 10(2):282–312, 1988.

[166] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *PLDI*, pages 78–88, 2007.

[167] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, pages 53–66, 2011.

[168] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-End Sequential Consistency. In *ISCA*, pages 524–535, 2012.

[169] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predictive Analysis for Detecting Serializability Violations through Trace Segmentation. In *MEMOCODE*, pages 99–108, 2011.

[170] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound Predictive Race Detection in Polynomial Time. In *POPL*, pages 387–400, 2012.

[171] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC*, pages 8–8, 2001.

[172] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[173] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *FSE*, pages 37–46, 2010.

[174] C. SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. 1992.

[175] W. N. Sumner, C. Hammer, and J. Dolby. Marathon: Detecting Atomic-Set Serializability Violations with Conflict Graphs. In *RV*, pages 161–176, 2012.

[176] H. Sung and S. V. Adve. DeNovoSync: Efficient Support for Arbitrary Synchronization Without Writer-Initiated Invalidations. In *ASPLOS*, pages 545–559, 2015.

[177] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism. In *ASPLOS*, pages 13–26, 2013.

[178] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*, pages 2–13, 2005.

[179] 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.

[180] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, pages 15–26, 2011.

[181] C. von Praun and T. R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

[182] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*, pages 115–128, 2003.

[183] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/FSE*, pages 205–214, 2007.

[184] L. Wang and S. D. Stoller. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In *PPoPP*, pages 137–146, 2006.

[185] L. Wang and S. D. Stoller. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE TSE*, 32:93–110, 2006.

[186] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *OSDI*, pages 281–294, 2008.

[187] J. Wilcox, P. Finch, C. Flanagan, and S. N. Freund. Array Shadow State Compression for Precise Dynamic Race Detection. Technical Report CSTR-201510, Williams College, 2015.

[188] B. P. Wood, L. Ceze, and D. Grossman. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*, pages 671–686, 2014.

[189] M. Xu, R. Bodík, and M. D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *PLDI*, pages 1–14, 2005.

[190] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers Reconsidered, Friendlier Still! In *ISMM*, pages 37–48, 2012.

[191] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why Nothing Matters: The Impact of Zeroing. In *OOPSLA*, pages 307–324, 2011.

[192] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA*, pages 261–272, 2007.

[193] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*, pages 19:1–19:11, 2013.

[194] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*, pages 221–234, 2005.

[195] M. Zhang, S. Biswas, and M. 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.

[196] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, pages 121–132, 2007.