System Support for Concurrent Software Reliability

Brandon Lucia

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2013

Reading Committee:

Luis Ceze, Chair

Mark Oskin

Daniel Grossman

Program Authorized to Offer Degree:
Department of Computer Science and Engineering

Abstract

System Support for Concurrent Software Reliability

Brandon Lucia

Chair of the Supervisory Committee:
Associate Professor Luis Ceze
Computer Science and Engineering

Parallel and concurrent software is more complex than sequential software because interactions between concurrent computations and the ordering of program events can vary across executions. This nondeterministic variation is hard to understand and control, introducing the potential for concurrency bugs. This dissertation addresses two challenges related to concurrency bugs, focusing on shared-memory multi-threaded programs. First, concurrency bugs are hard to find, understand, and fix, but debugging is essential to software correctness. Second, concurrency bugs cause schedule-dependent failures that degrade system reliability.

We develop two new concurrency debugging techniques based on statistical analysis and novel abstractions of inter-thread communication. These techniques isolate communications related to bugs and reconstruct failing executions. We show several hardware and software system designs that efficiently implement these techniques. We also develop two techniques for automatically avoiding schedule-dependent failures due to atomicity violations, a common concurrent program failure. We use specialized serializability analyses to identify code that should be atomic and system support to enforce atomicity. We implement these techniques with architecture and system support. Finally, we develop a mechanism for general schedule-dependent failure avoidance. We use a statistical analysis and leverage large communities of deployed systems to learn how to constrain executions to avoid previously seen failures. We show a software-only distributed system implementation that avoids real software failures with overheads low enough for production use.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables vii

Chapter 1: The Concurrent Software Reliability Problem 1
1.1 Concurrency and Parallelism in Shared-memory Multi-threaded Programs . . 3
1.2 Concurrency Bugs and Schedule-Dependent Failures 7
1.3 Challenges Addressed by this work 13
1.4 Contributions 17

Chapter 2: Bugaboo: Debugging with Context-Aware Communication Graphs . . 21
2.1 Context-Aware Communication Graphs 22
2.2 Implementing Context-Aware Communication Graph Collection 28
2.3 Debugging with Context-Aware Communication Graphs 30
2.4 Evaluation 31
2.5 Conclusions, Insights, and Opportunities 38

Chapter 3: Recon: Debugging with Reconstructed Execution Fragments 41
3.1 Reconstructed Execution Fragments 43
3.2 Debugging with Reconstructions 50
3.3 Implementation 55
3.4 Evaluation 59
3.5 Conclusions, Insights, and Opportunities 66

Chapter 4: Architecture Support for Context-Aware Communication Graphs . . . 68
4.1 CACG-HW: Architectural Support for Context-Aware Communication Graphs 69
4.2 Evaluation 73
4.3 Conclusions, Insights, and Opportunities 77

Chapter 5: Atom-Aid: Avoiding Atomicity Violations . . . 79
5.1 Background on Implicit Atomicity . . . 82
5.2 Implicit Atomicity Hides Atomicity Violations . . . 84
5.3 Actively Hiding Atomicity Violations . . . 86
5.4 Design Overview . . . 89
5.5 Implementing Atom-Aid with Implicit Atomicity and Hardware Signatures . 93
5.6 Evaluation . . . 96
5.7 Conclusions, Insights, and Opportunities . . . 105

Chapter 6: ColorSafe: Avoiding Multi-variable Atomicity Violations . . . 107
6.1 Multi-variable Atomicity Violations and Serializability . . . 108
6.2 ColorSafe: Detecting and Avoiding Multi-Variable Atomicity Violations . . . 110
6.3 Architectural Support . . . 115
6.4 Debugging with ColorSafe . . . 121
6.5 Evaluating ColorSafe . . . 122
6.6 Conclusions, Insights, and Opportunities . . . 130

Chapter 7: Aviso: Avoiding Schedule-Dependent Failures . . . 133
7.1 Schedule-Dependent Failures . . . 134
7.2 System Overview . . . 136
7.3 Monitoring Events and Failures . . . 140
7.4 Generating Constraints and Avoiding Failures . . . 143
7.5 Selecting and Distributing Constraints . . . 149
7.6 System Implementation . . . 155
7.7 Evaluation . . . 157
7.8 Conclusions, Insights, and Opportunities . . . 167

Chapter 8: Related Work . . . 170
8.1 Debugging Concurrency Errors . . . 171
8.2 Avoiding Schedule-Dependent Failures . . . 188

Chapter 9: Conclusions . . . 202
9.1 Cross-cutting Themes . . . 202
9.2 Final Thoughts . . . 204

Bibliography . . . 205

LIST OF FIGURES

Figure Number Page

1.1 **An illustration of a shared memory multi-threaded program execution.** The threads execute memory accesses (squares) that can read and write values to a shared memory space. Each thread executes its operations sequentially, but different threads' operations are independent and unordered by default. Threads can use synchronization operations (circles) to impose an ordering on operations in different threads that would otherwise be unordered. 4

1.2 **Different executions of a shared-memory multi-threaded program** . 6

1.3 **Code with a concurrency bug that can lead to an atomicity violation failure.** 10

1.4 **A concurrency bug involving multiple related variables that can lead to an atomicity violation.** 12

2.1 **High-level view of how communication graph structure can reveal a failure.** Markers represent memory operations involving shared data. . . . 23

2.2 **Debugging an atomicity violation example with a communication graph.** Different interleavings of the code in (a) yield different communication graph edges in (b). 24

2.3 **A basic communication graph is often insufficient for debugging.** Comparing edge sets from failing and non-failing executions' graph does reveal the cause of the failure. 25

2.4 **A context-aware graph reveals the cause of a multi-variable atomicity violation.** Different executions produce different sets of communication graph edges. Using a context-aware graph, edge differences reveal the communication responsible for the failure. Using a context-oblivious graph, it does *not*. 29

2.5 **Graph convergence.** Graphs reach a convergent structure with an increasing number of executions' graphs. The figure shows MySQL (a), Apache (b), and PBZip2 (c). 36

3.1 **Recon reconstructs fragments of program execution.** 41

3.2 **Overview of Recon's operation.** 42

3.3 **A buggy program and communication graph.** The dashed graph edge represents the buggy communication. 44

3.4 **A buggy program, a failing execution schedule, and its context-aware communication graph.** Nodes represent the execution of operations in a specific context. Edges represent communication between nodes. Note that we only include events in nodes' contexts that appear in our abbreviated trace for simplicity's sake. 45

3.5 **A timestamped communication graph and corresponding reconstruction.** The graph and reconstruction are based on the program, execution schedule, and graph in Figure 3.4. 47

3.6 **Aggregating reconstructions from many executions.** (a) shows reconstructions of many different failing program executions. (b) shows the resulting aggregate reconstruction with confidence values. 50

3.7 **Pair-wise feature plots illustrating class separation.** The plots show how effectively each pair of features separate reconstructions of the failure from others for `apache`. We only show the top 2000 ranked reconstructions and points representing reconstructions of the failure are circled. 54

4.1 **CACG-HW architectural extensions to a typical multiprocessor.** New architectural components are shaded. 70

5.1 **A simple example of an atomicity violation.** The read and update of `counter` from two threads may interleave such that the counter is incremented only once. 80

5.2 **Opportunities for interleaving.** (a) shows where interleaving from other threads can happen in a traditional system. (b) shows where such interleavings can happen in systems that provide implicit atomicity. 81

5.3 **Fine- (a) and coarse-grained (b) access interleaving.** There are six possible interleavings for the fine-grained system and two possible interleavings for the coarse-grained system. 83

5.4 **Naturally hiding an atomicity violation.** The figure shows the boundaries of a sequence of instructions intended to be atomic within dynamic atomic block boundaries. P_{hide} is the probability that the entire sequence executes within the block. 85

5.5 **Probability of hiding atomicity violations as a function of dynamic atomic block size.** 86

5.6 **Identifying data involved in a potential atomicity violation.** AtomAid discovers that `counter` might be involved in an atomicity violation and adds it to the *hazardDataSet*. 90

5.7 **Actively hiding an atomicity violation.** When `counter` is accessed, a block boundary is inserted automatically because `counter` belongs to the *hazardDataSet*. 91

5.8 **Block boundary insertion logic.** Flowchart showing Atom-Aid's policy for inserting dynamic atomic block boundaries. 93

5.9 **Signatures used by Atom-Aid to detect likely atomicity violations.** 94

5.10 **Empirically evaluating natural hiding.** Experimental data on the natural hiding of atomicity violations with implicit atomicity for various block sizes and bug kernels. Points show empirical data, curves show data predicted by our analytical model (P_{hide}). 100

5.11 **Atomicity violations hidden by Atom-Aid.** Results are averaged over all trials and error bars show the 95% confidence interval. 101

6.1 **Example atomicity violations.** (a) shows a single-variable violation and (b) shows a multi-variable violation. The example in (b) was distilled from https://bugzilla.mozilla.org/show_bug.cgi?id=73291. 109

6.2 **Unserializable color access interleavings.** 111

6.3 **Example of a unserializable color interleaving.** The example corresponds to case 5 in Table 6.2. `str` and `length` are left mutually inconsistent. 111

6.4 **Overview of how ColorSafe detects multi-variable atomicity violations.** The numbers in the dark circles denote the order of events happening simultaneously in (a) and (b). 113

6.5 **Keeping color access history.** 117

6.6 **Detecting unserializable interleavings in (a) debugging mode and (b) deployment mode.** In (a), only actual interleavings are being considered for the serializability test: the current access to a, the local history item i and the remote history items with $k \geq i$. In (b), all items in the remote history are being considered for the serializability test: local history item i, followed by local history item j, and all possible remote history items ($k = 0...n-1$). 120

6.7 **Violations avoided in bug kernels and full applications.** Results are shown for experiments using manual and malloc data coloring. †We used a different system configuration for MySQL. We explain the details in Section 6.5.2 (Difficulties with MySQL). 124

6.8 **Impact of history item granularity on violation avoidance.** The plot shows the number of atomicity violations avoided in kernel NetIO under synthetic noise for fine- and coarse-grain history items with a constant history window. 128

7.1 **A schedule-dependent failure in AGet-0.4.** 135

7.2 **The Avoidance-Testing Duality.** 137

7.3 **Aviso's components.** The compiler and profiler find and instrument events. The runtime system monitors events and failures and avoids events. The framework generates constraints, selects likely effective constraints using a statistical model, and shares effective constraints in a community of software instances. 138

7.4 **Enumerating pairs from a failing execution's RPB.** There are three threads, and time proceeds left to right. Circles are events, and arcs between events are event pairs. Arcs for duplicate pairs are omitted. The figure shows a single 10-event window of events, but selection occurs for all 10-event windows. 144

7.5 **Constraint Activation**. *Available constraints* are those that Aviso has made available to the execution. *Active constraints* are constraint instances that have been instantiated and can trigger delays. The large, central arrow signifies Thread 1 executing event B. To the left of the arrow there are no instances of the constraint (B, A); event B is its activation event, so when B is executed an instance of the constraint is added to the Active Constraints set (shaded cloud). Aviso records that Thread 1 is the instance's activator in the instance. 145

7.6 **How a constraint avoids a failure.** The constraint is shown in the cloud and is made from events B and A; when a thread executes B, the constraint is instantiated. When another thread executes A, it is delayed. The left side shows an execution snippet that can be viewed as both an atomicity violation and an ordering violation. 147

7.7 **A use-before-initialization failure from Transmission and the constraint that avoids it.** 148

7.8 **Aviso's statistical model.** The event pair model tracks feature values for each constraint. The failure feedback model tracks constraints' failure rates. The combined model is comprised of the other two, yielding a selection probability for each constraint. 153

7.9 **Aviso's improvement in reliability.** We show data for (a)Memcached, (b)Apache, (c)AGet, (d)PBZip2, and (e)Transmission. The x-axis shows execution time in number of trials – logical time ticks for servers, executions for standalone applications. We ran each program for 8000 trials. The y-axis shows the the number of failures that have occurred at a given point in time *on a log scale*. The top (black) curve shows the worst case: every execution is a failure. The middle (red) curve shows the reliability of the baseline, compiled and run completely without Aviso. The bottom (green) curve shows the reliability with Aviso. 160

7.10 **Characterizing Aviso's behavior.** 165

LIST OF TABLES

Table Number Page

2.1 **Bug workloads used to evaluate Bugaboo.** AV indicates an Atomicity Violation, OV indicates an Ordering Violation, and MVAV indicates Multi-Variable Atomicity Violation. 32

2.2 **Bug detection accuracy using Bugaboo.** We report the number of code point inspections required before the corresponding bug was found, the number in parenthesis show the number of distinct functions. Note that one inspection indicates that zero irrelevant code points needed inspection, since the bug was found on the first. Results are averaged over five trials. 33

2.3 **Debugging effectiveness for BB-SW (word) with different context sizes.** Dash (—) indicates the bug was not found with the corresponding context size. 35

2.4 **Characterization of BB-SW and communication graphs sizes.** 37

3.1 **Effectiveness of features.** The table shows the reliefF rank of each feature for our C/C++ benchmark programs. 55

3.2 **Buggy programs used to evaluate Recon.** We used both C/C++ programs and Java programs and we included a variety of bug types. 60

3.3 **Properties of reconstructions for our benchmarks.** 61

3.4 **Performance of Recon.** We shows Recon's base configuration and many less-optimized configurations relative to uninstrumented execution. 64

4.1 **Bug detection accuracy using CACG-HW.** We report the number of code point inspections required before the corresponding bug was found. The number in parenthesis show the number of distinct functions. Note that one inspection indicates that zero irrelevant code points needed inspection, since the bug was found on the first. Results are averaged over five trials. 74

4.2 **Imprecision for different configurations of CACG-HW.** We show imprecision for line-level and cache-to-cache-only tracking of inter-thread communication. 75

4.3 **Characterization of CACG-HW.** 76

5.1 **Serializability analysis.** The table shows the analysis and interpretation of each interleaving described in [80]. 88

5.2 **Cases when an address is added to the *hazardDataSet*.** 92
5.3 **Bug benchmarks used to evaluate Atom-Aid.** 98
5.4 **Characterizing Atom-Aid.** The table shows data for both the signature and non-signature implementations. 103
5.5 **Characterization of the bug detection process for real applications using Atom-Aid.** 105
6.1 **Bugs used to evaluate ColorSafe.** 123
6.2 **Characterization of Ephemeral Transactions.** The rate of ET starts, % of useful ETs, and % of conflicting useless ETs for full applications in deployment mode. Ap2.0 and MySQL were run using malloc coloring, and AGet, manual coloring. MySQL was run with the modified configuration described in Section 6.5.2 (Difficulties with MySQL). 124
6.3 **Failure avoidance for a variety of ET sizes.** Applications marked with a ^{m} were run using manual coloring, because their bugs involve global and heap variables; All others were run with malloc-coloring. 127
6.4 **Impact of noise on ET usefulness.** Percentage of useful ETs in NetIO with synthetic noise, 12,000-instruction total history length, and varied history item granularity. 128
6.5 **Evaluation of ColorSafe for debugging.** Number of code points reported by ColorSafe using deployment mode, debugging mode, and debugging mode with invariant post-processing. 130
7.1 **Aviso's runtime overheads.** These overheads are relative to baseline execution when collecting events only and when collecting events and avoiding failures. 161
7.2 **Aviso's dynamic behavior.** Columns 2 and 3 show the total number of sharing events and the number of sharing events discarded due to online pruning. Columns 4 and 5 show the total number and number discarded of synchronization and signaling events. Column 5 shows the number of times an event in an available constraint executes, requiring a check to see if the event activates the constraint. Column 6 shows the number of times a check actually leads to a constraint's instantiation. Column 7 shows the number of times an event is delayed by a constraint. 166
8.1 **Categories of prior work discussed in this chapter.** 170

ACKNOWLEDGMENTS

This dissertation is the culmination of years of work that I have done under the guidance of incredible mentors who have led me to where I am from my naive beginnings. First and especially, I need to thank my advisor, Luis Ceze. I lucked out working with Luis. His effervescent attitude and creativity have shaped who I am as a researcher and a person. Luis helped to guide my intuitions and taught me how to really look at, and understand things. In advising me, Luis has put up with a lot from me, which probably required more patience than I realize. He has also fueled my wanderlust, by sending me around the world to preach about the research we've done together. Thanks for showing me the ropes, it really means a lot.

I have also appreciated Mark Oskin's irreverence and cynicism – well-applied, these yield the freshest insights. His mentorship has helped me calibrate my personal and professional worldview. Dan Grossman's vulcan lucidity and ability to focus on the essential has helped me to occasionally think before I speak — this ability may come in handy. Susan Eggers's advice and experience wrangling undergraduate students has been instructive and I value having taught alongside her. I appreciate Mike Ernst's candor, clarity, and willingness to help me improve my research and exposition.

Looking further back, I am reminded of how I ended up doing research to begin with. Soha Hassoun at Tufts University led me to my first steps down this path. Without her influence, I am unsure I would even know that the world of research exists. It was good times staring down boolean constraint systems for hours in the lab in Halligan Hall. Sam Guyer, too, helped to guide me toward research while I was at Tufts. I never did get around to finishing that object inlining code, but working together was enriching, nonetheless.

Looking yet further back, to past mentors from before my time at Tufts is difficult because it feels like eons have passed. Several stand out. Ed Dignum gave me self-confidence

and his mentoring trained me to be a speaker. Anthony Cassale saw promise in my interest in computers early on and I can't thank him enough for challenging me to write Pac-Man in Pascal. Todd Koza showed me Linux for the first time and, probably without knowing it, changed my life.

It is a true privilege to have had such guidance from such incredible mentors.

Aside from my mentors, the student members of SaMPA have also been professionally and socially enriching during my formative academic years and I must thank them all for ideas, feedback, and inspiration. Especially: To Joe Devietti for being one of the most balanced people I know. And of course, for finishing your PhD first; To Adrian Sampson for giving me something to aspire to when I'm making a talk and for lots of good times at conferences. To Jacob Nelson for your empathy, ability to make the cluster work, and often surprising knowledge of very specific subjects. To Nick Hunt, for help writing under fire and for having the self-awareness to follow your ambitions. To Ben Wood for your spirit of adventure and awesome connectionist sense of humor, immortalized multiple times in the proceedings of PoCSci. To Tom Bergan for your candid and constructive way of looking at things, and your good sense as a real engineer. To Hadi Esmaeilzadeh, for commiseration over job search gripes when we were both figuring out where we were headed.

In addition to my labmates, I must thank the cohort of others with whom I have worked during my time as a PhD student. It was a pleasure to collaborate on research with Hans Boehm, Karin Strauss, Shaz Qadeer, Laura Effinger-Dean, Emily Fortuna, and Todd Schiller.

Apart from my colleagues, the people in my life deserve gratitude that would fill far more than the space on this page. Especially: To Nicki for companionship, tolerance, support, encouragement and sense of belonging. Your perspective on life helps me to keep in mind what is important and to not take life too seriously. You make me a happy person. To Tony Fader, for being a great friend, unpretentiously intellectual, and a source of humor and inspiration. To Matt D'Arcangelis for being a great friend, for temperingly looking outward, helping me not to forget who I am. To Morgan, Franzi, Greg, Nell, Cliff, Todd, Pete and

everyone else for being wonderful friends over the years. For games nights, barbecues, happy hours, post-TGIF fun, *etc.*, that happily perforated the work-time of grad school. To the Racer Sessions and the people that make it special, for keeping me creative and amongst others who are even more creative. I also must nod to my recent past lives because they put me where I am now: Julia Verplank, the Quigleys (Katie and Rich), Bop Street Records, and The Foghorns. Life outside of grad school facilitated life inside grad school and all of these things are valuable to me.

Finally and especially, I am grateful beyond anything I could write for my family's unflagging support for my entire life and throughout my education. My parents influence and encouragement to seek education is what led me to my ambitions. Summers in Fonda, NY gave me the work ethic and perspective I needed for a PhD. You were always there when I needed you whether I realized it at the time or not. To my siblings: it's been inspiring that we're all so individual. It's often a relief to hang out with you guys.

And to all the people in my life that I've forgotten, I appreciate you too.

DEDICATION

To my parents and siblings for making me who I am

—and—

To the people in my life for being who you are.

Der Mensch kann tun was er will; er kann aber nicht wollen was er will.

— **Arthur Schopenhauer**

Chapter 1

THE CONCURRENT SOFTWARE RELIABILITY PROBLEM

The rate of improvement in performance of sequential computation resulting from device scaling has begun to lag behind the scaling trend projected by Moore's Law [95]. A recent study [42] attributes this change to the failure of device physics to follow the scaling projections made by Dennard [35]. As a result, system designers have turned to *parallelism* as the primary means to increase the performance of programs. Most modern, commercially available computer systems have multiple processor cores. These parallel computing resources may be within a multicore chip, within a system across several chips, or even spanning several different machines, as in a warehouse-scale or datacenter computer. To take advantage of parallel computer systems, programmers must write programs that orchestrate computations that can run in parallel. Exposing parallel work to parallel computing resources ensures resources remain utilized, resulting in efficient, high-performance computation. In addition to the need for parallel software to utilize parallel computation resources, there is a similar need by many applications for *concurrency*. Concurrent computations are logically simultaneous computations that may share resources. For example, cloud and server applications must coordinate communication and resource sharing by simultaneous client requests. Regardless of whether concurrent computations are executed in parallel or multiplexed on a single computing resource, sharing and communication require concurrency control.

A goal in writing software – sequential, concurrent, and parallel – is that it should be written correctly, (without programming errors or "*bugs*") and must execute reliably (without "*failing*"). Software has grown essential to the function of the world, controlling infrastructure, facilitating communication, disseminating information, and acting as a platform for the global economy. When software has bugs that cause it to fail, these essential roles

that it plays are interrupted. Sometimes, such interruptions have grave consequences [10] or cost stakeholders millions of dollars [11]. NIST estimates the cost of preventable software bugs to be a non-negligible fraction of the GDP of the US [99].

Making software correct and reliable is difficult. Writing, testing, and debugging software is a challenge that has been studied for years in academia and industry. A large fraction of this effort has been devoted to making sequential software correct and reliable. A key barrier to reliability is that software often has a vast space of possible states, many of which are dependent on the computation's input or the environment in which the software is executing. The state space that emerges, given a program's structure, possible inputs, and possible execution environments is highly complex. A programmer must consider all the possible program states and reason that each behaves according to the specification of the software. Ensuring that this is the case is usually done through some combination of careful code-writing, judiciously applied testing, trial and error debugging, and various development tools. The complexity of the state space is what makes it challenging to create correct, reliable sequential software.

Concurrency and parallelism exacerbate the difficulty of making correct, reliable software. As in sequential software, in concurrent and parallel software, the state space is in part defined by the program's structure, the possible inputs, and the environment in which the program may execute. With concurrency and parallelism, however, the space is *fundamentally more complex*. The often extremely large number of possible interactions between concurrently executing computations increases the size of the space. The manner in which computations are carried out in parallel – scheduling, resource allocation, and parallelization strategy – also impact the size of the state space of the program.

The fundamental increase in complexity for concurrent and parallel software as compared to sequential software presents several challenges to programmers. It is more difficult to write programs that correctly execute computations in parallel. It is a challenge to correctly coordinate the sharing of resources by concurrent computations. It is difficult to find, understand, and fix errors related to parallelism and concurrency. The increased complexity of the state space of concurrent and parallel programs makes testing substantially more complicated. The difficulty in testing means that some bugs will not be found and may

surface only in production systems, causing those systems to fail.

The work in this dissertation focuses on the correctness and reliability of concurrent and parallel programs. This work develops novel techniques that simplify the process of debugging concurrent programs and that make software execute reliably in production *despite* latent bugs in deployed code. These techniques are realized through the use of novel computer architecture and system support implemented in both the hardware and software layers of a computer system. A high-level statement of the thesis of this work follows:

> **Thesis Statement:** Novel system and architecture support addresses the complexity of concurrent and parallel software by making it easier to debug and by making it execute without failing, despite bugs in code.

The remainder of this chapter is devoted to providing context for the work in this dissertation. First, we describe shared-memory multi-threaded programming, the model of computation around which we conducted this work. Second, we describe the types of bugs and failures addressed in this work. Third, we discuss the main challenges addressed in this work. Fourth, we describe each of the contributions of this dissertation in brief, foreshadowing later chapters that describe them in detail.

1.1 Concurrency and Parallelism in Shared-memory Multi-threaded Programs

While there are many different models for writing concurrent and parallel software (*e.g.*, [20, 7, 6, 8, 2]), one of the most common is *shared-memory multi-threaded programming*. The work in this dissertation focuses on shared-memory multi-threaded programs. Unlike sequential programs that have a single thread of control, multi-threaded software has *multiple* threads of control. Each thread executes its own instructions in some sequence. Threads interact by sharing data and synchronizing. To share data, threads perform *read* and *write* operations that load values from and store values to locations in a shared address space. Operations performed by different threads are unordered by default: there are no implicit constraints on the order of execution of different threads' operations. Using *synchronization* [57, 72, 5], a programmer can impose an order on two operations in different threads.

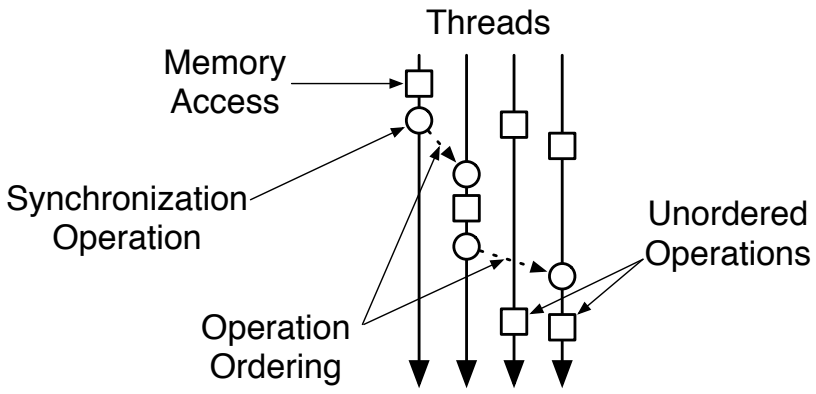


Figure 1.1: **An illustration of a shared memory multi-threaded program execution.** The threads execute memory accesses (squares) that can read and write values to a shared memory space. Each thread executes its operations sequentially, but different threads' operations are independent and unordered by default. Threads can use synchronization operations (circles) to impose an ordering on operations in different threads that would otherwise be unordered.

When two operations are ordered in an execution by executing in the same thread or by synchronization, the first is said to have *happened before* the other. The ordering imposed is referred to as the "happens-before order" [71] for that execution and it is an irreflexive partial order on the events in the execution. Figure 1.1 illustrates several aspects of a shared-memory multi-threaded program's execution.

As a program executes, threads observe an *execution schedule* of their and other threads' memory and synchronization operations that corresponds to the execution's happens-before order. The execution schedule shows the order of operations in an execution and determines which read operations in the program read values written by which write operations. There are two important factors that determine the observed execution schedule for a particular execution.

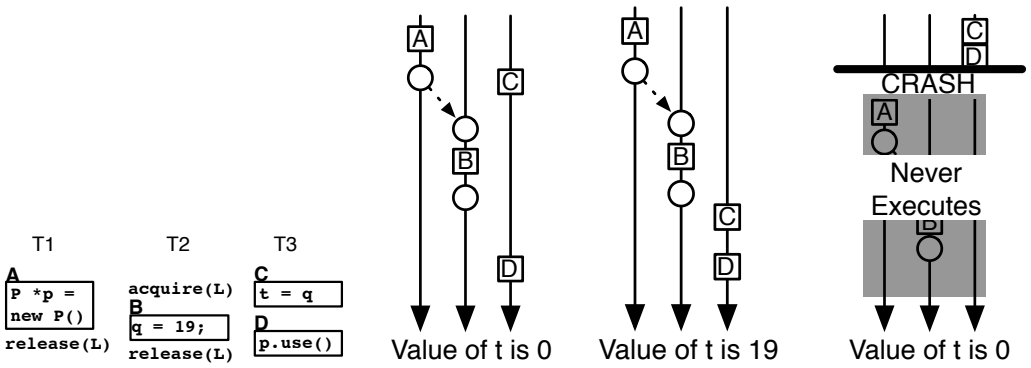
- **Synchronization Order**: The order in which synchronization operations are executed may vary. The function of synchronization operations is to determine the order of other operations in a program's execution. Synchronization operations themselves, however, do not have a fixed *a priori* order at the program's start. Differences in the timing of operations in different executions can alter the order in which threads reach

synchronization points. The variation in the order in which threads arrive at synchronization points can in turn vary the order of other operations in the execution. These variations imply that different executions may have different execution schedules.

- **Data Races:** Happens-before is a partial order over the operations in an execution, meaning some operations are *not* ordered before or after one another. These operations are *concurrent*. A pair of concurrent operations *conflict* if they both access the same piece of shared state (*i.e.*, the same variable) and at least one of the operations modifies the state (*i.e.*, performs a write operation). The concurrency of non-conflicting, concurrent operations in an execution schedule is inconsequential: such operations do not access similar state or synchronize, so they cannot have any effect on one another. In contrast, the execution of two concurrent, conflicting operations constitutes a *data-race*. The execution semantics of racy operations is tricky: the accesses conflict, so their execution order may affect their outcome, but the accesses are concurrent, so their order – and thus their outcome – is unclear. The outcome of a data-race can vary nondeterministically from one execution to the next. Data-races are a major challenge. Section 1.2.1 describes the mechanics of data-races and the problems they present in more detail.

The execution schedule of a program can vary from one execution to the next because of data-races and variation in the order of synchronization. That variation implies that there are many, different, possible, valid execution schedules for a program. A given execution may follow any of those schedules – the schedule that an execution follows is *nondeterministic* from one execution to the next. Nondeterminism is an important property of shared-memory multi-threaded programs because *different execution schedules can lead to different results, even with the same input.*

Figure 1.2 illustrates how operations can execute in a different order from one execution to the next, assuming an arbitrary memory model to resolve the order of racy operations. Figure 1.2(a) shows the pseudocode of a simple program with three threads, T1, T2, and T3. T1 is assigning a shared pointer, `p`, to point to a new object and then releasing a lock.



(a) **A shared-memory multi-threaded program.** Thread 1 assigns `p` to a new object. Thread 2 assigns `q` to 19. Thread 3 reads `p` and `q`.

(b) **One possible execution schedule.** Thread 2's read of `q` in **D** sees that `q = 0` because it executes before **B**.

(c) **Another possible execution schedule.** Thread 2's read of `q` in **D** sees that `q = 19` because it executes after**B**.

(d) **A failing execution schedule.** Thread 3's attempt to use `p` in **D** causes the program to crash, because when **D** executes, Thread 1 has not yet initialized `p`.

Figure 1.2: **Different executions of a shared-memory multi-threaded program**

T2 acquires the lock, assigns a new value to a shared variable, `q`, and then releases the lock. T3 reads the value of `q` into a local variable, `t`, and then uses `p` by dereferencing it. `p` is assumed to be uninitialized at the start of the execution. `q` is assumed to be `0` at the start of the execution. Figure 1.2(b) shows one execution, leading to one result state. Figure 1.2(c) shows a different execution leading to a different result state.

Figure 1.2(a-c) illustrates that variation in the execution schedule can vary the result of a computation. Figure 1.2(d) shows how variation in the execution schedule can even determine whether a computation *fails or succeeds*. The operations in **C** and **D** are not ordered with the operations in **A**. When **C** and **D** occur in the execution schedule without observing the result of the code in **A**, the pointer dereference in **D** dereferences `p` before it has been initialized by the assignment in **A**. Assuming, as many languages do [4, 24, 88], that dereferencing an uninitialized pointer is illegal, such a schedule leads to a failure. The synchronization that the programmer wrote is *buggy* and permits the execution schedule shown in part (d) that leads to a failure.

In most programs, most execution schedules behave as the programmer intended. How-

ever, as Figure 1.2(d) shows, when the programmer has written buggy code, that code may permit schedules that cause the program to do something that it was not intended to do (like crash). The bug in Figure 1.2(a) is a *concurrency bug*. Concurrency bugs are errors in code that can lead to *schedule-dependent failures*. A schedule-dependent failure is any unintended program behavior that is the result of particular orderings of operations in different threads. These failures are called "schedule-dependent" because their occurrence depends on the occurrence of execution schedules that exhibit those particular orderings. Concurrency bugs and the schedule-dependent failures they cause are two fundamental barriers to correctness and reliability in concurrent programs.

1.2 Concurrency Bugs and Schedule-Dependent Failures

In shared-memory multi-threaded programs, concurrency bugs are errors in code that coordinates inter-thread interactions, like ordering operations and sharing data. Concurrency bugs show up as incorrectly used synchronization operations and patterns of accesses to memory shared by multiple threads that lead to unintended behavior. There are many types of concurrency bugs that lead to schedule-dependent failures [79]. The next sections discuss some of the most common types, including *data-races*, *atomicity violations*, *ordering violations*, as well as some important variations on these classes of errors.

1.2.1 Data-Races

No discussion of concurrency errors would be complete without a discussion of data-races. A data-race occurs when two threads each perform an access to the same memory location, at least one of the threads' accesses is a write, and the accesses are not ordered by synchronization (*i.e.*, happens-before). Many popular programming language definitions ascribe executions of programs with data-races undefined [4, 24, 5] or unintuitive [88] semantics. The intent of these semantics is to permit aggressive instruction reordering optimizations in compilers and architecture, without excessive restriction from the language. Despite weak or overly complex language-level semantics, most computer systems execute programs with data-races and the ordering semantics of racy operations is determined by the architecture

level memory model [109, 120, 9, 52]. An architecture's memory model determines, for each point in an execution, the value that a read (racy or otherwise) of shared state may observe.

Data-races lead to variation in the execution schedule. All data-races are not necessarily schedule-dependent failures, but many are and the behavior of data-races is hard to reason about. Some in the research community take the strong position on data-races that, because language-level semantics and architecture-level memory models make reasoning about data-races so difficult, any code that permits a data-race is buggy and no data-race is benign [23].

Why are data-races so hard to reason about? The root of the difficulty presented by data-races is that programmers typically reason about their programs assuming a memory model under which all executions are *Sequentially Consistent* or *SC* [71]. In an SC execution the same order of operations is observed by all threads and threads' operations execute in their original program order. Unfortunately, most architecture memory models do not guarantee SC in all circumstances. Instead, they provide some form of *relaxed consistency*. Relaxed consistency allows systems to use aggressive optimizations that can reorder a thread's operations and cause threads to observe different execution orders. In executions with no data-races, these reorderings cannot have any effect on the behavior of the program. When a program has data-races, however, reorderings involving racy operations can change the program's behavior from one execution to the next. Reordering of racy operations can also result in unintuitive behavior that violates SC and causes a program's execution to apparently defy causality. [1]

The work in this dissertation does not explicitly address SC violations that stem from data-races — other work by this author [84, 38] and plentiful work by other authors (*e.g.*, [45, 141, 89, 90]) addresses this dimension of data-races. Instead, this dissertation treats data-races as sub-sequences of the execution schedule that can vary from one execution to the next, sometimes leading to failure behavior. These variations may lead to violations of SC, or may be a source of nondeterminism only. In either case, if the reorderings lead to behavior that violates the specification of the program, then the code involved in the data-race constitutes

[1] A detailed discussion of memory models, reordering, and consistency is outside the scope of this dissertation. A gentle introduction to the subject can be found in this tutorial [12] and in this primer [124].

a concurrency bug. Other types of concurrency bugs discussed in this section may involve operations that race.

1.2.2 Atomicity Violations

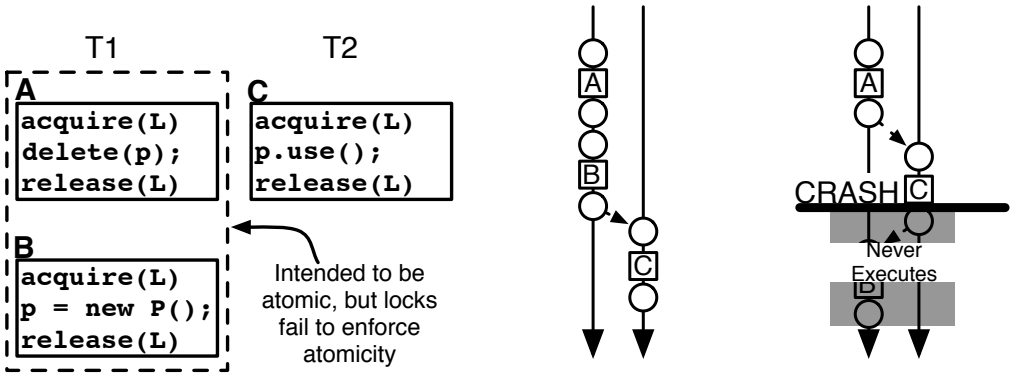
Atomicity violations are another common type of concurrency error. Critical to understanding atomicity violations are the concepts of *Atomicity* and *Isolation*.

- **Atomicity** is a property of a sequence of program operations. A sequence of operations are atomic only if all of their results become observable by other threads simultaneously. Other threads may observe none of the results of the atomic instructions or all of the result of the atomic instructions, but never the result of some and not of others.

- **Isolation** is a property of a sequence of program operations. A sequence of operations are isolated if the result of their execution concurrently with code in other threads is the same as the result of their execution in the absence of other threads. Other threads may not observably access the same state as an isolated computation while it is executing, nor may the isolated computation observably access the same state as the other threads' computations.

Programmers often intend for a sequence of operations to be atomic and isolated. The atomicity and isolation of a region of code can be ensured with proper use of synchronization. An execution of a program with a concurrency bug that omits or incorrectly uses synchronization to enforce atomicity may fail due to an *atomicity violation*. An atomicity violation is a type of failure that can afflict a region of code intended to be atomic and isolated. If such a region accesses some shared memory locations and its execution is *interleaved* by instructions in another thread that access the same locations, the interleaving may violate the atomicity of the region of code.

Figure 1.3 shows how an interleaving permitted by a concurrency bug can lead to an atomicity violation. The code in Thread 1 deletes and subsequently reallocates the data



(a) **A program with a concurrency bug.** The bug permits an atomicity violation, leading to a failure.

(b) **An execution schedule without an atomicity violation.** Thread 2's dereference of `p` correctly uses the reallocated object.

(c) **An execution schedule with an atomicity violation.** Thread 2's dereference of `p` happens between `delete` and `new`, *violating the atomicity of those operations*. The atomicity violation leads to a failure.

Figure 1.3: **Code with a concurrency bug that can lead to an atomicity violation failure.**

pointed to by `p`. Thread 1's code should execute atomically. When Thread 2's code *interleaves* between Thread 1's manipulations of `p`, Thread 2 dereferences a deleted pointer, leading to a crash. The code in Thread 1 that should have been atomic is implemented incorrectly. A single locked critical region should contain the delete and the reallocation, not two separate regions.

Some interleavings with regions of code intended to be atomic lead to atomicity violations. The interleavings that lead to atomicity violations are *unserializable*. Serializability [102] is a property of a concurrent execution. A concurrent execution is serializable if its result is the same as some sequential execution of the regions of code in the execution that were intended to be atomic. If a region of code was intended to be atomic and the outcome of that region is *changed* simply because some other code interleaved between the operations in that region, then the interleaved execution was unserializable. Put another way, when an interleaving can change an execution's behavior, it is unserializable. Atomicity violations stem from unserializable interleavings of regions of code intended to be atomic. Prior work

has shown [44, 48, 80, 136] that if an execution is serializable (with respect to some specified atomic regions), then there were no atomicity violations.

The connection between atomicity violations and serializability makes serializability a useful property. *Serializability analysis* facilitates reasoning about the difference between atomicity constraints that were implemented by a programmer (*e.g.*, using locks), and those that were specified in the program's design (*e.g.*, in a specification). However, the problem of atomicity violations runs deeper than simply checking the serializability of specified atomic regions: Often there is no correct specification of the regions of code in a program that were intended to be atomic. The absence of atomicity specifications has two effects: (1) Programmers often incorrectly reason about which code should be atomic and isolated. They use synchronization incorrectly, and write broken code that suffers atomicity violations. (2) Detecting atomicity violations requires inferring atomic code regions, which is difficult.

Bugs leading to atomicity violations are very common. The work in this dissertation develops several techniques for debugging such bugs and avoiding atomicity violations.

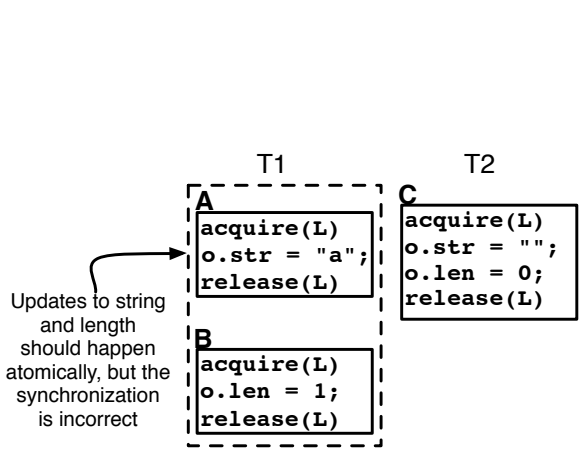
1.2.3 Ordering Violations

Ordering violations are another common class of concurrent programming mistakes [79]. Ordering errors consist of a pair of operations that should execute in a particular order, but for which synchronization constraints are absent, permitting them to execute out of order. Some ordering violations involve accesses that constitute a data-race. Other ordering violations involve only synchronized accesses, although in these cases, the synchronization does not enforce the correct order of the operations, only that there is no data-race. To prevent failures due to ordering violations, programmers must use synchronization to ensure that the correct event order occurs in all executions. The bug in Figure 1.2(a) leads to an ordering violation that involves racy accesses to `p` in **A** and **D**.

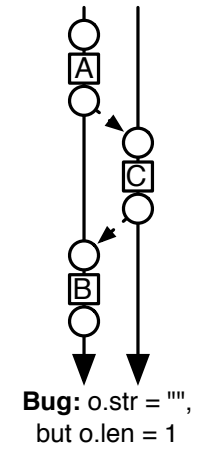
1.2.4 Multi-variable Concurrency Errors

There are several variants of each of the types of concurrency errors discussed so far. One important axis along which the errors can vary is the amount of data referred to by oper-

ations involved in the error. Prior work has primarily characterized errors as being either *single-variable* or *multi-variable* errors [14, 78, 80, 55, 83, 81, 86]. The operations making up single-variable errors all access the same memory location; the operations making up multi-variable errors may access several different memory locations.



(a) **A program with a concurrency bug involving multiple variables.** `o.str` and `o.len` represent related properties of a string. When one is update, the other should also be updated and the updates should be atomic. Thread 1 is executing buggy code that updates both properties, without atomicity. Thread 2 is also updating both properties



(b) **An execution schedule with an atomicity violation.** Thread 2's update of the variables interleaves between Thread 1's. At the end of both threads, the variables' values are not consistent with one another.

Figure 1.4: **A concurrency bug involving multiple related variables that can lead to an atomicity violation.**

Data-races are inherently single-variable errors because a data-race is defined as a pair of accesses to the same memory location (although some prior work has expanded the notion of a data-race to consider races at a coarser data-granularity [129]). Atomicity violations and ordering violations can involve either a single variable or multiple variables. Multi-variable variants of atomicity and ordering errors are more complex than their single-variable counterparts. The complexity arises from the need to consider not just interactions between threads involving a single piece of state, but rather, arbitrary subsets of the program's state (*i.e.*, groups of variables).

Figure 1.4(a) illustrates a multi-variable concurrency bug. The code in the figure has

two threads. Both threads are attempting to update two related properties of an object, `o` that represents a string's contents (`o.str`) and its length (`o.len`). The properties are semantically related to one another. If one property is updated, the other must be updated to maintain the consistency of their relationship. Furthermore, the updates must happen atomically. Thread 2 correctly uses synchronization to make its updates the variables atomic. However, Thread 1 uses synchronization incorrectly, failing to ensure the atomicity of its updates. Figure 1.4(b) shows an execution of this program in which Thread 2's code violates the atomicity of Thread 1's code. When that happens, the result of the two updates inconsistent: `o.str` is the empty string (`""`) and `o.len` is 1.

1.2.5 Other Concurrency Errors

The list of errors in this section is not comprehensive. Instead, this list has focused on errors addressed by the work in this dissertation. There are other types of concurrency errors, such as *deadlocks* [31] and *livelocks*, which prevent executions from making any progress, as well as errors involving thread interactions via shared resources other than shared memory [69]. The work in this dissertation is applicable to some deadlocks and livelocks, but does not address errors that involve accesses to resources other than shared memory.

1.3 Challenges Addressed by this work

There are two goals to the work in this dissertation: (1) to simplify the process of finding, understanding, and fixing concurrency bugs; and (2) to develop techniques that automatically avoid schedules that lead to schedule-dependent failures, even in programs with concurrency bugs that permit those schedules. Concurrency debugging and schedule-dependent failure avoidance are essential challenges presented by concurrent and parallel software.

1.3.1 Debugging Concurrency Bugs

The goal of programmers is to find the bugs in their programs and to fix those bugs. Programmers test their programs and use bug-finding tools to uncover indicators of bugs in their code, like bug reports and failed test cases. Starting from those indicators, program-

mers must then understand the bug and determine what code changes are necessary to fix it. There are several reasons why debugging concurrency bugs is especially difficult.

1. **Complexity of execution schedule space.** The execution schedule can vary from one run to the next nondeterministically. The space of possible execution schedules that a program may exhibit is very large. It is often not obvious from code alone which execution schedules may lead to failure, or even which schedules are possible in the first place.

2. **Implicit inter-thread interaction.** Threads in a shared-memory multi-threaded program interact by reading and writing to shared memory. Unfortunately, it is often unclear whether a memory operation accesses shared or thread-local state: *inter-thread interaction is implicit.* Implicit inter-thread interaction makes it hard for a programmer to reason about which code might be involved in a concurrency bug.

3. **Non-local reasoning.** A program operation that accesses shared state or synchronizes has an impact not only on the thread executing the operation, but potentially on all other threads in the program. A schedule-dependent failure may occur as the result of such an operation. When such a failure occurs, it is not explicit which other threads and which other code might have been involved in that failure. Concurrency bugs involve code in different threads. Understanding a schedule-dependent failure that manifests at a point in one thread's code requires reasoning "non-locally" about code executing in one or many other threads. Non-local reasoning makes understanding and fixing concurrency bugs difficult.

The work in this dissertation directly addresses these challenges, using novel techniques and novel system and architecture support to make concurrency debugging simpler.

This work addresses the challenge posed by the very large space of possible thread schedules by incorporating and analyzing information from *many* executions. Using novel abstractions and analyses, these techniques determine what parts of the execution schedule are invariant in failing or non-failing executions. Using this information, the techniques in

this dissertation focus a programmers' attention on the parts of a program that are likely to be related to a schedule-dependent failure.

This work also addresses the challenges posed by implicit inter-thread interaction and non-local reasoning by developing new abstractions that encode inter-thread interaction through shared memory. The explicit encoding of inter-thread interaction highlights the parts of the code in different threads that interact. Furthermore, the abstraction shows "both ends" of an interaction between two threads – aiding in understanding local and non-local effects of an operation. Making implicit interactions explicit and taking a global view of thread interaction simplify concurrency debugging.

1.3.2 Avoiding Schedule-Dependent Failures

Schedule-dependent failures are the result of concurrency bugs. When concurrency bugs are not corrected before software is deployed to a production system, schedule-dependent failures will degrade the reliability of that system. Concurrency bugs are especially problematic because comprehensive testing for concurrency bugs is infeasible using existing methods [96, 25]. Lacking comprehensive testing, software is often released with *latent* concurrency bugs that only manifest as failures in production. To remain reliable, systems must execute programs despite these latent errors, avoiding execution schedules that manifest schedule-dependent failures. The work in this dissertation is among the first to study mechanisms that make systems automatically avoid schedule-dependent failures. There are several essential challenges systems face in avoiding schedule-dependent failures.

1. **Absence of correctness specifications and failure criteria.** Programs are frequently written without an explicit specification that describes what their behavior should be. In the absence of a specification, it is unclear what defines a failure. As a result of the absence of explicit specifications, systems must infer when a failure has occurred.

2. **Need to intervene before a failure occurs.** Techniques for avoiding failures must intervene before a failure actually causes a problem. The need to be preemptive re-

quires a system to infer when a failure is likely, *before it has happened*. Inference is difficult because non-failing execution schedules often closely resemble failing execution schedules.

3. **Preserving program semantics.** A systems must change some aspect of a program's execution (*e.g.*, the execution schedule) to avoid an imminent failure. Altering a program's execution presents a risk of doing more harm than good, if the alteration can lead to new and potentially unintended (*i.e.*, failing) program behavior. Alterations to an execution to avoid a failure should not change the executing program's semantics.

4. **Use in production systems.** To remain reliable, systems must avoid failures due to latent bugs *in production*. Working in production puts stringent bounds on the acceptable performance impact of these techniques. Production performance requirements limit the amount of analysis possible and the permissible cost of failure avoidance actions.

The techniques for avoiding schedule-dependent failures described in this dissertation overcome these problems.

One of the main contributions of this dissertation is to show that strategically perturbing a program's execution schedule avoids many failures *without altering program semantics*. The intuition behind this contribution is that many different schedules are valid and perturbing away from a failing schedule toward a non-failing schedule avoids a failure and remains a correct execution.

The techniques in this work use novel analyses to infer when failures may happen. Some of the analyses in this work infer likely failures by leveraging bug-specific properties. Other analyses work by analyzing information from many failing and non-failing executions to determine what behavior is most likely indicative of an impending failure. Both varieties of analysis presented do not require explicit specifications of correct behavior or failure behavior.

As required, the failure avoidance mechanisms and the analyses described in this dissertation work within the narrow permissible performance window of a production system.

1.4 Contributions

This dissertation tackles the challenges outlined in the previous section with novel system and architecture support. The mechanisms developed in this dissertation simplify concurrency bug debugging and enable systems to avoid schedule-dependent failures, demonstrating the main thesis of this work.

1.4.1 Architecture and System Support for Debugging Concurrency Errors

This dissertation describes three concrete contributions that use system and architecture support to simplify the process of debugging concurrency bugs. These three systems are described in brief here. Chapters 2, 4, and 3 describe them in detail.

- **Bugaboo** [81] We develop a new abstraction for a concurrent program execution called the *Context-Aware Communication Graph*. Context-Aware Communication Graphs encode inter-thread interactions via shared memory. A graph node represents the execution of a program instruction in a particular *communication context*. Communication context abstractly encodes a partial history of accesses to shared memory preceding the instruction. We develop a statistical debugging methodology using context-aware communication graphs and show it is useful for debugging. We describe a low-complexity, software-based, reference implementation of a tool for collecting context-aware communication graphs. We show that these graphs provide important information that simplifies concurrency bug debugging in real software.

- **Recon** [86] We build a debugging methodology based on *reconstructed execution fragments* or reconstructions. Reconstructions are short, time-ordered sequences of communicating instructions derived from context-aware communication graphs (from Bugaboo). We build a statistical model of reconstructions from a set of executions, representing each reconstruction with a number of numeric-valued features. We use a

simple rank inference heuristic that uses these features to determine which reconstructions are most likely related to the failure being debugged. We describe an efficient system implementation for Java and for C/C++ programs. Our results show that Recon precisely pinpoints information that is useful for debugging with few distractions. We also show that our software implementations have overheads that are comparable to other widely used debugging tools.

- **Architecture Support for Context-Aware Communication Graphs** [81] We develop hardware architectural support for collecting context-aware communication graphs. Our architecture support uses existing hardware support for cache coherence, along with some simple meta-data extensions to caches and a small amount of additional buffering. We show that the context-aware communication graphs collected using our architecture support are useful in a technique such as Bugaboo. We also show that the complexity of our design is reasonable and it eliminates the software overheads associated with collecting context-aware communication graphs. Finally, we discuss the use of the architecture extensions in deployed systems to continuously collect communication graphs from production.

1.4.2 Architecture and System Support for Avoiding Schedule-Dependent Failures

This dissertation describes three concrete contributions that use system and architecture support to automatically avoid schedule-dependent failures. These three systems are described in brief here. Chapters 5, 6, and 7 describe them in detail.

- **Atom-Aid** [85] We observe that systems with implicit atomicity [27, 32, 132] naturally prevent some schedule-dependent failures due to atomicity violations and characterize the effect. We develop *Atom-Aid*, which uses implicit atomicity support and serializability analysis to infer "hazardous" data, likely to be involved in atomicity violations. We adaptively insert atomic block boundaries to enclose groups of accesses to hazardous data, to avoid atomicity violations. We show that reporting code where inferred block boundaries occur identifies code that is likely to be related to an atom-

icity violation. We describe a possible system implementation based on an execution model from prior work that provides implicit atomicity [27]. We show our technique can avoid failures in real programs without any software overheads or excessive architectural complexity beyond our base design.

- **ColorSafe** [83] We extend Atom-Aid's analysis to handle multi-variable atomicity violations. We apply "colors" to groups of data, giving the same color to related data. We develop a "color-space" variant of atomic-set serializability analysis [55] to infer sets of same-colored hazardous data that are likely to be involved in an atomicity violation. We develop an analysis that, guided by the set of hazardous data colors, identifies sequences of code that should be atomic. We use transactional memory support to enclose groups of accesses to hazardous data in the same transaction, making them atomic. We show that inferred atomic sequences are also helpful in pointing out where bugs may exist in code. We show that ColorSafe avoids failures due to multi-variable atomicity violations in real programs. We describe an architecture implementation that does not require support for implicit atomicity and thus has lower complexity than Atom-Aid. We show that our implementation does not impose software overheads and has only modestly high implementation complexity.

- **Aviso** [82] We develop a software-only system for automatically avoiding schedule dependent failures. Aviso monitors events during a programs execution. When a failure occurs, Aviso records a history of events from the failing execution. It uses this history to generate schedule constraints that perturb the order of events in the execution to avoid schedules that lead to failures in future program executions. Aviso leverages scenarios where many instances of the same software run, using a statistical model of program behavior and experimentation to determine which constraints most effectively avoid failures. We show that Aviso decreases failure rates for a variety of important desktop, server, and cloud applications by orders of magnitude, with an average overhead of less than 20% and, in some cases, as low as 5%.

This dissertation continues by proceeding with a chapter discussing each of these contributions. Chapters 2 – 4 focus on concurrency bug debugging. Chapters 5 – 7 center on schedule-dependent failure avoidance. Each of these chapters includes a terminal section entitled "Conclusions, Opportunities, and Insights". Each one of these sections recapitulates the contributions of its chapter, describes its chapter's most important concepts and insights, and describes the limitations and opportunities for future work on its chapter's subject. Following the chapters that discuss the contributions of this dissertation is a summary of related prior research in Chapter 8. Chapter 9 concludes with a brief statement of cross-cutting themes that show up in the work in this dissertation and some final thoughts.

Chapter 2

BUGABOO: DEBUGGING WITH CONTEXT-AWARE COMMUNICATION GRAPHS

Concurrent programming errors often manifest as interactions between threads that programmers did not anticipate. The main mode of interaction between threads in shared-memory multi-threaded programs is for threads to read and write data to parts of the memory space that they share. When threads share data through memory, those threads are *communicating*. Inter-thread communication occurs when one thread writes a value to shared memory and another thread reads or overwrites that value. *Communication graphs* are a class of graph abstractions that can represent the inter-thread communication that took place in a multi-threaded program execution. In the simplest form of communication graph, nodes represent static memory operations and edges represent inter-thread communication between pairs of static memory operations. The pattern of inter-thread communication that occurs during a program's execution and the corresponding communication graph directly result from the order of operations in the execution's schedule. As discussed in Chapter 1, each different execution can exhibit a different schedule and, as a result, a different corresponding communication graph. The main hypothesis of this chapter is that executions leading to schedule-dependent failures include some characteristic inter-thread communication. Communication graphs from failing executions therefore differ in structure from graphs corresponding to non-failing executions.

We leverage our main hypothesis in a technique to identify and understand concurrency bugs. Given some communication graphs corresponding to non-failing program executions and some others corresponding to failing program executions, we compute the differences between the failing and non-failing executions' graphs. Looking at these graph differences focuses on what is essential to the failure behavior and atypical of non-failing program behavior, which is helpful for understanding and fixing the underlying bug. The biggest

advantage of our graph-based approach to debugging is that it is *general*: it does not rely on heuristics that are specific to a class of concurrency bugs. Using communication graphs is useful for finding any failure that stems from an unintended pattern of inter-thread communication, which includes most concurrency bugs.

A fundamental challenge we face in designing such an approach, however, is developing a satisfactory communication graph abstraction. This task presents two conflicting design constraints. Our abstraction must include *as much information as is necessary* to differentiate between failing and non-failing executions' communication behavior. At the same time, our abstraction must also include *as little information as is sufficient*, remaining amenable to online collection in efficient space and time. In this chapter, we develop the *context-aware communication graph*. Context-aware communication graphs are a communication graph abstraction that strikes a balance between these design goals. The guiding insight of our context-aware communication graph design is that nodes represent a *dynamic instance* of a particular static code point, as it executes in a particular *communication context*. The *communication context* of a node abstractly encodes inter-thread communication operations preceding the execution of the instruction represented by the node. Context-aware communication graphs are discussed in detail in Section 2.1.

In this chapter we introduce *Bugaboo* a new technique and system design that is useful for debugging multi-threaded programs. Bugaboo makes several key contributions. First, we describe context-aware communication graphs. Second, we propose a debugging technique that uses context-aware communication graphs to find the code related to a schedule-dependent failure. Third, we describe a simple implementation of Bugaboo's context-aware communication graph collection mechanism, which uses software support only. Finally, we evaluate our techniques and show that they are useful for debugging a variety of complex concurrency bugs, with a modest programmer effort.

2.1 Context-Aware Communication Graphs

Our approach to concurrency bug debugging begins with a bug report that describes a schedule-dependent failure and an input that sometimes leads to that failure. A programmer using our technique runs the program repeatedly with the failure-triggering input. Bugaboo

collects inter-thread communication graphs from each execution. Some executions fail and others do not; the programmer annotates each graph as having come from either a failing or a non-failing execution. Bugaboo then analyzes the collected graphs to detect differences between the failing and non-failing graphs that are likely to be related to the failure. The key idea is that presence or absence of certain edges distinguish a correct execution's graph from an incorrect execution's graph. In this way, graph differences directly reflect buggy communication.

Figure 2.1 illustrates this concept. Figure 2.1(a) shows an event ordering bug taken from MySQL-4.1.8. Figure 2.1(b) shows the communication graphs obtained from a correct (top) and an incorrect execution (bottom). In the incorrect execution there is no communication between the store to `dynamicId` in Thread 1 and the load in Thread 2, so Thread 2 reads uninitialized data. Comparing these graphs directly points to the communication that is characteristic of the failing execution. We discuss graph processing in more detail in Section 2.3.

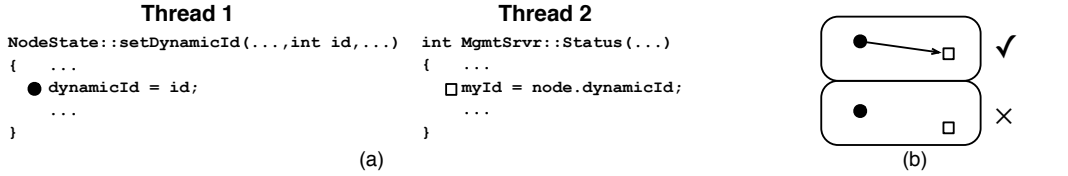


Figure 2.1: **High-level view of how communication graph structure can reveal a failure.** Markers represent memory operations involving shared data.

2.1.1 Context-Oblivious Communication is Insufficient for Concurrency Debugging

A key part of using communication graphs for debugging is deciding what information is encoded by our chosen graph abstraction. A very simple communication graph abstraction represents static code points with nodes and communication between static code points via shared memory with edges. We call this very simple abstraction a "context-oblivious" communication graph. Context-oblivious graphs do not encode any information about which

dynamic instance of a particular static code point participated in communication. Generally, we call information that distinguishes one dynamic instance of a code point from another the "context" of that dynamic instance. One of the main contributions of this chapter is to show that a context-oblivious communication graph does not encode sufficient information for a general approach to concurrency bug debugging.

In a context-oblivious communication graph, schedule-dependent failures may lead to graph edges that are present only in failing executions' graphs; therefore simple graph differences might point to buggy code points. For example, consider the bug in Figure 2.2(a), extracted from MySQL: if the read access of `log_type` in Thread 2 interleaves between the two writes in Thread 1, a failure occurs. Figure 2.2(b) shows a communication graph resulting from a union of graphs from a failing and a non-failing execution. The bad interleaving in (a) leads to the dashed edge in (b) that only appears in failing executions.

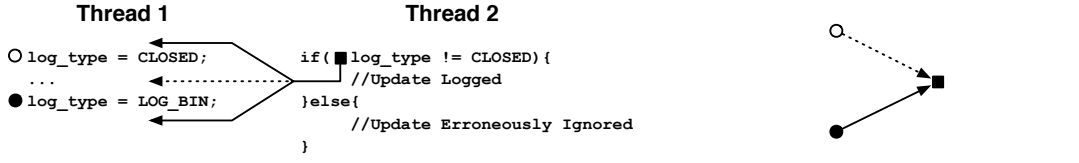


Figure 2.2: **Debugging an atomicity violation example with a communication graph.** Different interleavings of the code in (a) yield different communication graph edges in (b).

As illustrated by Figure 2.2, context-oblivious graphs can be used to debug some concurrency bugs, but they cannot be used to debug many others. This is because graphs from executions exhibiting schedule-dependent failures often contain edges that are also present in graphs from non-failing executions. Figure 2.3(a) shows a multivariable atomicity violation, in which two variables representing a string and its length are not updated atomically. The read accesses of `str` and `len` in Thread 2 will get inconsistent data if they interleave with the write accesses in Thread 1 and cause an atomicity violation. However, as Figure 2.3(b) shows, this interleaving does not lead to a unique edge in the communication graph. The lack of any difference between the failing and non-failing executions' graphs makes it impos-

sible to isolate any communication event as being characteristic of the failure using graph differences.

To recapitulate, the edges in the communication graph that are the result of the atomicity violation are also present in a graph from a non-failing execution. The context-oblivious communication graph is lacking because it does not distinguish between different *dynamic instances* of the code points involved in this failure. In Figure 2.3, the dynamic instances of the code points in Thread 2 that occur between the two operations in Thread 1 are characteristic of the failure. In contrast, the dynamic instances that both occur either before or after the operations in Thread 1 are indicative of correct behavior.

(a)
Multi-variable atomicity violation example. Arrows represent a few possible interleavings. Dashed arrows represent buggy interleaving.

(b)
Communication graph. Dashed edges represent buggy communication. Edges alone can not distinguish correct/incorrect communication.

Figure 2.3: **A basic communication graph is often insufficient for debugging.** Comparing edge sets from failing and non-failing executions' graph does reveal the cause of the failure.

2.1.2 Adding Communication Context to Communication Graphs

One way of distinguishing between different dynamic instances of a static code point is by making nodes directly represent dynamic memory operations. Such a "fully dynamic" communication graph distinguishes between *all* different dynamic instances of every static code point. A key design concern in this work is to ensure that communication graph collection is time and space efficient. Fully dynamic communication graphs are effectively traces of all memory operations from an execution. Such a graph structure is impractical because its size is *unbounded*, growing with execution time. Graph size that scales with execution length fails to satisfy our space-efficiency design constraint. We propose *context-aware commu-*

nication graphs, a new communication graph abstraction that distinguishes between *some* different dynamic instances of communicating code points, but remains bounded in size and is not significantly larger than a context-oblivious communication graph.

The key aspect of a context-aware communication graph is that nodes represent a combination of static memory instruction and the *communication context* in which the instruction executed. The communication context of a memory instruction is the sequence of potentially communicating memory instructions observed by the executing thread immediately prior to the execution of the memory instruction. We obtain each memory instruction's communication context by monitoring potentially communicating memory operations that are observed by a thread. The communication context is an abstract encoding of these observed operations. We encode memory operations that make up the context as *context events* by discarding the data and instruction address of the operations, keeping only the operation type.

There are four types of context events:

1. *LcRd*, a read of data recently written by a remote thread

2. *LcWr*, a write to data recently read or written by a remote thread

3. *RmRd*, a remote read of data recently written locally

4. *RmWr*, a remote write to data recently read or written locally

As a thread observes context events, it inserts them into a context queue, which is a fixed-size FIFO. When a thread executes a communicating memory operation that must be added to the context-aware communication graph, the contents of the thread's context queue is the communication context of that memory operation's graph node. Context size is arbitrary, and the longer it is, the more ordering information is encoded in the graph.

An important property of communication context is that context events are collected for any potentially communicating memory operations, regardless of its instruction or data address. With this property, communication context abstractly encodes the *global order*

of inter-thread communication operations – precisely the information that is important for distinguishing between failing and non-failing patterns of communication.

Formalism and Properties of Context-aware Communication Graphs Formally, a context-aware communication graph is defined as $G = (V, E)$, where $v \in V$ is a tuple $(inst, ctx)$, and each edge $(u, v) \in E$ is a pair of these tuples. An edge $((u.inst, u.ctx), (v.inst, v.ctx))$ is present in G if during the execution from which G was constructed, two conditions held:

1. **Condition 1:** a thread executed $u.inst$ when that thread's context queue contained $u.ctx$ and wrote a value to a shared memory location, x

2. **Condition 2:** a different thread than the one in Condition 1 executed $v.inst$ when that thread's context queue contained $v.ctx$ and read or overwrote the value written to x by $u.inst$ in Condition 1

.

Context-aware communication graph nodes distinguish between some different dynamic instances of static instructions, which are directly represented as nodes in context-oblivious graphs. Each node in a context-oblivious communication graph therefore maps to multiple nodes in a context-aware communication graph from the same execution. If the context-oblivious communication graph of an execution has N_s nodes, a context-aware communication graph of the same program execution will have at most $N_s \times C$ nodes, where C is number of all possible communication contexts in which a memory access can execute. Since there are four types of events, $C = 4^S$, where S is the context size. We experimentally determined (Section 2.4.3) that a context of five events (1024 possible contexts) is enough to capture enough ordering to detect all types of concurrency bugs discussed in the literature. In practice, the addition of context does not mean that a context-aware communication graph is 1024 times bigger, because each node executes in a small fraction of possible contexts. Our experiments (Section 2.4.6) never showed an increase larger than 50-fold.

Figure 2.4 shows how context-aware communication graphs can be used to debug concurrency bugs. Figure 2.4(a) shows multiple executions of the buggy code in Figure 2.3(a).

For each execution, it shows the executed sequence of memory operations. The symbols in parenthesis represent the communication context of each thread at the point when the communication happened. For example, refer to the first execution (top) in Figure 2.4(a): the first operation on the left (write to `str`) was executed after two remote reads had been observed by the local node, so the context at that point is (RmRd, RmRd); the context becomes (RmRd,RmRd,LcWr) after that operation when the next operation on the left is executed (write to `len`). This sequence of context updates illustrates how the context adds an abstract encoding of the global operation order to the communication graph.

Figure 2.4(b) shows that, unlike a context-oblivious communication graph, a context-aware communication graph makes it possible to identify the failure-inducing communication that occurs between the write to `str` and read from `str`. Recall that context-oblivious communication graphs are limited in exposing concurrency bugs because there are no edges that are only present in the graphs of incorrect executions. With context-aware graphs there are edges that are only present in graphs of incorrect executions.

2.2 Implementing Context-Aware Communication Graph Collection

At the algorithmic level, context-aware communication graph collection is a dynamic analysis. Each thread tracks its current communication context. As memory operations execute and communication occurs, communication graph edges are added to the graph.

We describe a software implementation of Bugaboo, BB-SW, that is appropriate for use on commodity multiprocessors with modest run time and storage overheads. BB-SW was implemented using the Pin binary instrumentation framework. BB-SW instruments memory accesses with calls to our runtime that monitor those memory accesses and build the communication graph. Our runtime has three key data-structures: (1) a *last writer table* that maps each memory location to an entry containing the instruction address, thread ID, and communication context of the last writer. This table can be configured to track memory locations at word or line granularity; (2) a per-thread array that implements each thread's context queue; and (3) a graph data-structure that stores the context aware communication graph itself.

Whenever a thread reads from or writes to a memory location, the thread checks the

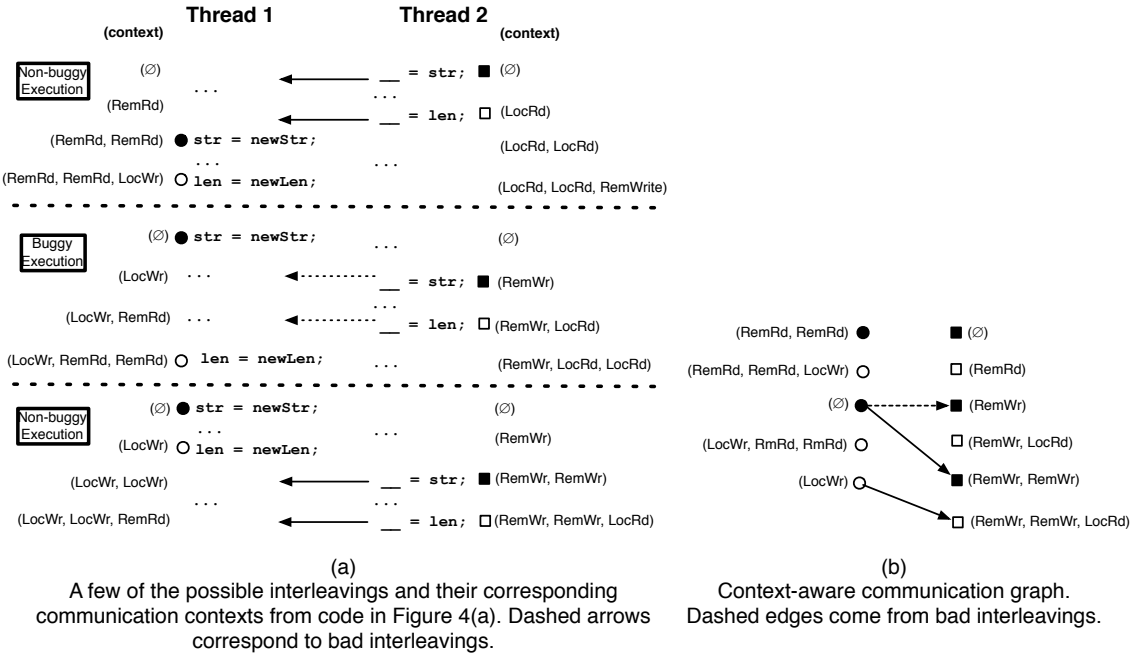


Figure 2.4: **A context-aware graph reveals the cause of a multi-variable atomicity violation.** Different executions produce different sets of communication graph edges. Using a context-aware graph, edge differences reveal the communication responsible for the failure. Using a context-oblivious graph, it does *not*.

last writer table to determine which thread last wrote that location. If the last write was performed by a different thread, the thread executes code to add a new edge to the graph. The source node of the new edge is created with the last writer's instruction address and context. The edge's sink node is created with the thread's executing instruction and the contents of the thread's context queue.

To maintain the communication context, whenever a thread accesses a location last written by another thread, it records a corresponding LcRd or LcWr event identifier in its context queue. The thread that last wrote that location records a corresponding RmRd or RmWr event in its context. The size of the event FIFO queue is fixed at five and when full, the oldest element is discarded.

2.3 Debugging with Context-Aware Communication Graphs

We develop a method for concurrency bug debugging with context-aware communication graphs. Using our debugging technique, the programmer collects a large set of communication graphs and annotates each as having come from a *failing* or *non-failing* program execution. Bugaboo then compares the set of failing graphs to the set of non-failing graphs and reports a list of edges that distinguish failing executions from non-failing ones.

Collecting and Labeling Graphs

The first step in this debugging methodology is for the programmer to run the application multiple times using BB-SW, collecting a communication graph from each execution. The programmer then labels each graph as having come from a failing or non-failing execution. The process of collecting failing and non-failing graphs can be assisted by testing tools that attempts to force bugs to happen [96, 105]. Our labeling process assumes that the programmer classifies an execution as failing if the failure being debugged manifests, regardless of whether other failures manifest.

Analyzing Graphs and Ranking Code Points

Once graphs have been collected from many executions and labeled, our analysis produces a set of *failure-only* graphs. There is a failure-only graph for each graph from a failing execution. A failing execution's failure-only graph contains the set of edges that occur in that failing execution's graph, but not in any non-failing execution's graph.

Computing the failure-only graphs cuts down the set of edges under consideration substantially, compared with the original set of graphs from failing executions. However, each failure-only graph contains two classes of edges: (1) edges in the graph because they are related to the cause of the failure; and (2) edges in the graph because there is a large diversity of program behavior and many edges occur only in failing program executions simply by chance. Our tools goal is to isolate the first kind of edges (those related to the bug) and to de-emphasize the second kind of edges (those in the graph by chance). To accomplish this goal, we assign a *rank* to each of the code points in the failure-only graphs.

To compute a code point's rank, we rely on a prior assumption about the distribution of different contexts in which a code point executes. We assume that a code point tends to execute in a small, invariant set of contexts when that code point is not involved in causing a failure. Our rank function looks for instances of a code point that deviate from this assumption, executing in failing executions, in an unusual context. Such code points receive a higher rank than those that adhere to this assumption.

To find such code points, we analyze each edge in each failure-only graph. We first break each analyzed edge into its two constituent code points. The rank of a code point, CP, is defined as: $rank_{CP} = \sum_{x \in X_{CP}} \frac{F_{CP,x}}{F_{CP,*}}$. X_{CP} is the set of contexts that occur in nodes with code point CP in any failure-only graph. $F_{CP,x}$ is the number of failure-only graphs in which code point CP exists in a node with context x. $F_{CP,*}$ is the total number of graphs (failure-only and non-failing) in which CP occurs in a node, regardless of context and across all runs. We sort the list in ascending rank order. In Section 2.4.2 we demonstrate that this method is very effective at detecting concurrency errors, despite a few irrelevant code points that get a high rank.

2.4 Evaluation

This evaluation aims to: (1) demonstrate that our debugging methods based on context-aware communication graphs detect bugs accurately, leading to few unnecessary code inspections; (2) characterize size and accuracy of our graphs; (3) characterize the behavior and performance of BB-SW.

2.4.1 Experimental Setup and Methodology

We conducted our experiments using our full implementation of BB-SW, described in Section 2.2. We used three categories of workloads: full applications, bug kernels, and synthetic buggy code. Table 2.1 shows the workloads used. The full applications were chosen based on previous literature on bug detection [79, 85, 139]. To exercise buggy code in Apache, we enabled buffered logging, and used a custom script which launched 10 simultaneous requests for a static resource. For our experiments with MySQL, we enabled binary logging, and used the included sql-bench utility, modified to execute 50 instances of the test-insert

benchmark in parallel. For PBZip2, we decompressed a bzip compressed text file containing a communication graph from our tool. For AGet we fetched a software archive from a remote server, and interrupted the download with the Unix interrupt signal. In AGet and PBZip2, we added Unix `usleep` calls to more frequently cause the bug to manifest itself. Our bug kernels were extracted from Mozilla and MySQL. They are 300-600 line extracts including buggy code from these applications. We used bug kernels to capture the essence of bugs, and make in-depth experimental analysis less cumbersome. This methodology has been used successfully in prior work in this area [80, 85, 139]. Finally, we used several synthetic bug benchmarks. Several of these were used in prior work on atomicity violation detection [80, 85], and we added a synthetic ordering violation bug.

	Name	Version	Type	Description
Synthetic	BankAcct	n/a	AV	Two threads try to update a bank account balance simultaneously, and an update is lost.
	CircularList	n/a	AV	Many threads remove elements from head of queue and append them to tail of queue. Lack of atomicity of remove/append leads to incorrect append order.
	LogAndSweep	n/a	AV	A log is written by many threads, and periodically flushed. Missing atomicity constraint leads to log corruption.
	MultiOrder	n/a	OV	Two threads' repeated accesses to a shared variable must be interleaved. No code constraint enforces interleaving.
Bug Kernel	Moz-jsStr	0.9	MVAV	To compute avg. string length, total number of strings and total string length are tracked. Non-atomic updates can permit these to become inconsistent
	Moz-jsInterp	0.8	MVAV	Cache data structure is populated, and flag indicating cache occupancy is set. Lacking atomicity constraints, interleaving read may read flag while it is inconsistent with cache.
	Moz-macNetIO	0.9	MVAV	Read of "valid" flag in conditional test and outcome of conditional can be interleaved, and data invalidated.
	Moz-TxtFrame	0.9	MV	During update of buffer offset and buffer text length variables, inconsistent values can be read by interleaving read.
	MySQL-IdInit	4.1.8	OV	Query of database node ID should be ordered with assignment of node ID but absent ordering constraints lead to incorrect ID in query reply.
Full App.	MySQL-BinLog	4.0.12	AV	Attempts to log data during log rotation do not properly handle log being closed, leading to unlogged database transactions.
	Apache-LogSz	2.0.48	AV	Concurrent updates to length of text in buffer can cause dropped update, leading to corruption of buffer. Can lead to crashes and log corruption.
	PBZip2-Order	0.9.1	OV	Termination of worker thread loops is not ordered with deletion of pthread cond. var. data structure. Accesses to deleted cond. var. causes crash.
	Aget-MultVar	0.4	MVAV	Value of shared var. should be consistent with # bytes written to output file. Lacking atomicity constraint permits read of inconsistent value of shared var. in signal handler.

Table 2.1: **Bug workloads used to evaluate Bugaboo.** AV indicates an Atomicity Violation, OV indicates an Ordering Violation, and MVAV indicates Multi-Variable Atomicity Violation.

2.4.2 Efficacy

We applied the debugging methodology described in Section 2.3 to each benchmark program. Recall that the output of BB-SW is a rank-ordered list of code points, the first of which is most likely related to the failure being debugged. We measure the quality of the output by the number of code points ranked higher than the first code point that is part of the

bug, i.e., the *number of inspections* required before the bug is found. All results presented are averaged over 5 trials. For each trial, we collected graphs from 25 failing runs and 25 non-failing runs. We justify the number of runs in Section 2.4.5. Table 2.2 lists each bug, whether we were able to detect it with and without context (Columns 2-3), and the number of code point and function inspections required to find the bug (Columns 4-5).

Overall, our results demonstrate that our technique accurately pin-points concurrency errors, even in very large software packages. We isolated code points related to the bug in each benchmark with only a few code inspections. In many cases, a code point related to the bug was the *first code point reported*.

Benchmark	**Debuggable *without* Context**	**Debuggable *with* Context**	**# of Code Inspections Required**	
			BB-SW *Line* Granularity	**BB-SW *Word* Granularity**
BankAcct	No	Yes	4.0 (1.0)	3.6 (1.4)
CircularList	No	Yes	1.2 (1.0)	2.6 (1.2)
LogAndSweep	No	Yes	1.0 (1.0)	1.0 (1.0)
MultiOrder	No	Yes	1.0 (1.0)	1.0 (1.0)
Moz-jsStr	No	Yes	1.0 (1.0)	1.0 (1.0)
Moz-jsInterp	No	Yes	2.6 (1.6)	1.8 (1.0)
Moz-macNetIO	No	Yes	5.0 (3.6)	3.0 (1.6)
Moz-TxtFrame	No	Yes	1.8 (1.4)	3.4 (1.0)
MySQL-IdInit	Yes	Yes	1.0 (1.0)	1.0 (1.0)
MySQL-BinLog	No	Yes	28.4 (19.2)	34.2 (21.2)
Apache-LogSz	No	Yes	8.8 (7.2)	13.2 (10.8)
PBZip2-Order	No	Yes	10.8 (2.6)	6.6 (2.0)
AGet-MultVar	No	Yes	1.0 (1.0)	1.0 (1.0)

Table 2.2: **Bug detection accuracy using Bugaboo.** We report the number of code point inspections required before the corresponding bug was found, the number in parenthesis show the number of distinct functions. Note that one inspection indicates that zero irrelevant code points needed inspection, since the bug was found on the first. Results are averaged over five trials.

2.4.3 The Importance of Communication Context

Column 2 (*Detected without Ctx*) and Column 3 (*Detected with Ctx*) in Table 2.2 show that, with just one exception, we can debug the bugs in our evaluation using context-aware communication graphs, but not context-oblivious ones. The exception is an ordering violation bug: `MySQL-IdInit`. We can debug this bug without context because there is a pair of memory accesses that communicates only during failing runs. For the other bugs, without context, there were no context-oblivious communication graph edges in graphs from

our experimental executions that occurred only during failing executions.

Our dependence on context does not mean other techniques will not find these bugs. For example, AVIO can detect some of the atomicity violations that require context (e.g., `BankAccount`) because AVIO uses a heuristic specific to atomicity violations. At the time of the initial publication on this work [81], however, we were unaware of another approach that was useful for debugging the multivariable atomicity violations in Table 2.1.

2.4.4 Detecting Bugs With Context-Aware Communication Graphs

Columns 4, and 5 in Table 2.2 show the number of code point inspections to find a code point related to the bug. In Bugaboo's ranked list of reports, there is a code point related to the bug for all of the programs in our evaluation. Most experiments required few unnecessary code inspections and, in some cases, none at all. The application requiring the most inspections was `MySQL-BinLog`, with approximately 34 (in 21 different functions), which is a reasonable number considering that the code consists of over one million lines. For `Apache-LogSz`, which has over 220k lines of code, we only needed to look at 8.8 code points on average to find code related to the bug. For `Aget-MultVar`, a smaller application with less than 5k lines of code, there was never any code point ranked higher than code related to the bug.

Comparing Columns 4 and 5 shows the effect of tracking last writer meta-data at the granularity of a word (*i.e.*, 64 bits), versus tracking it at the granularity of a cache line (*i.e.*, 64 Bytes). Comparing these results shows that there is little change in debugging precision with the change in meta-data granularity. This result suggests that tracking last writer meta-data at cache line granularity is a good idea, requiring less total storage for meta-data and reducing interference between program data accesses and meta-data accesses in the cache. This result also shows that tracking last writer meta-data is amenable to implementation in hardware, using per-cache-line last writer meta-data. We discuss such a hardware implementation in Chapter 4.

Sources of Irrelevant Reports

There are two main reasons irrelevant code points are sometimes highly ranked. First, nondeterministic multi-threaded execution may lead to potentially rare, but correct communication. If sufficiently rare, or if only ever observed in failing executions, these may be ranked highly, in spite of being correct. Second, because failure behavior tends to be infrequent, when buggy code executes, the resultant communication context might also be infrequent with respect to subsequent communicating instructions not involved in the bug. This rare communication can lead to these involved instructions also having a rare context and appearing as bugs in our ranking. Both of these data sparsity problems can be mitigated by using communication graphs from more program executions.

2.4.5 Effect of Context Size

Benchmark	None	1-Entry	2-Entry	3-Entry	4-Entry	5-Entry
BankAcct	—	2.0	2.0	2.4	3.4	3.6
CircularList	—	—	7.2	3.4	3.2	2.6
LogAndSweep	—	—	2.2	2.8	1.6	1.0
MultiOrder	—	—	2.8	1.0	1.0	1.0
Moz-jsStr	—	1.0	1.0	1.0	1.0	1.0
Moz-jsInterp	—	—	—	1.6	1.8	1.8
Moz-macNetIO	—	1.2	1.0	1.4	2.2	3.0
Moz-TxtFrame	—	—	3.4	2.8	2.8	3.4
MySQL-IdInit	1.0	1.0	1.0	1.0	1.0	1.0
MySQL-BinLog	—	—	522.6	128.8	60.0	34.2
Apache-LogSz	—	25.2	6.4	7.2	9.8	13.2
PBZip2-Order	—	—	—	—	6.6	6.6
AGet-MultVar	—	3.8	3.8	1.4	1.0	1.0

Table 2.3: **Debugging effectiveness for BB-SW (word) with different context sizes.** Dash (—) indicates the bug was not found with the corresponding context size.

Table 2.3 shows how Bugaboo's bug detection ability varies with the context size. Note that some bugs cannot be found with context size lower than 4 (`PBZip2-Order`). For `MySQL-BinLog`, the number of inspections required to find the bug is high unless a longer context is used. For most applications, as context grows, the number of irrelevant inspections goes down, which is expected since more ordering information is available to distinguish memory accesses involved in bugs. We chose Bugaboo's default context size to be 5 because

we wanted to favor better bug coverage and lower unnecessary inspections even if at the cost of increasing graph size.

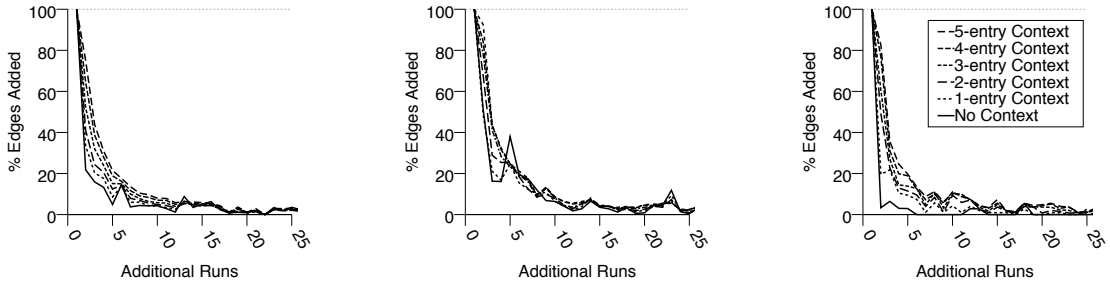


Figure 2.5: **Graph convergence.** Graphs reach a convergent structure with an increasing number of executions' graphs. The figure shows MySQL (a), Apache (b), and PBZip2 (c).

We now show that after few program runs, the number of unobserved communication graph edges in each additional run decreases rapidly, i.e., we obtain a *convergent* communication graph quickly. Figure 2.5 shows the number of new communication graph edges observed during each program execution as a fraction of the total size of the union of graphs from all prior executions. As expected, with longer contexts, more executions are necessary to reach a convergent graph.

The set of observed edges converges, as indicated by the sharp drop-off in new communication graph edges, after about 10-15 executions. The data show that as the length of the context increases, the number of executions required to converge increases as well. The increase occurs because as the context gets longer, each static instruction corresponds to a larger set of nodes (*i.e.*, one node for each distinct context). At a few points in the sequence of executions, the fraction of edges added is *greater* for smaller context sizes. This apparent inversion occurs because graphs with longer contexts are larger, and the percent increase in edges contributed by a particular execution is lower for larger graphs. These results show that the number of executions required to collect a convergent graph is proportionate to the length of the context, and that for any context size, very few executions are necessary. This justifies the choice of 25 runs for our evaluation, since it is sufficient for convergence.

2.4.6 Characterization

Our characterization has two goals: to assess the performance cost of BB-SW; and to measure the typical size of context-aware communication graphs. We did this characterization using the full applications from Table 2.1 as well as the PARSEC [17] benchmark suite, since synthetic bugs and bug kernel executions lack adequate diversity for such a characterization.

Benchmark	BB-SW Slow-down(x)	Graph Sizes			
		w/o Context		w/ Context	
		# Nodes	# Edges	# Nodes	# Edges
blackscholes	128	51	104	230	472
canneal	80	216	437	2025	4055
dedup	451	227	750	3784	11570
ferret	26	398	821	572	1216
fluidanimate	4623	284	831	15692	38570
freqmine	3845	1050	2228	41455	85142
swaptions	2151	168	676	5103	15633
vips	5025	1326	2942	56016	115178
x264	1260	2347	4799	68067	137071
AGet-MultVar	15	58	135	154	376
PBZip2-Order	19	59	145	208	451
Apache-LogSz	13	672	1361	1797	3635
MySQL-BinLog	166	1303	3271	20435	48861

Table 2.4: **Characterization of BB-SW and communication graphs sizes.**

Overheads of BB-SW

Column 2 of Table 2.4 shows the slowdown caused by BB-SW compared to the application running natively, without any instrumentation. As expected, BB-SW causes significant performance degradation, because each memory operation requires a call into the runtime to update the communication graph and each thread's context queue. The cost of the action varies depending on how frequently inter-thread communication occurs in the application. For some applications, such as `Apache-LogSz` and `AGet`, we saw tolerable slowdown of about 15x. For some applications (e.g., `Vips`), the cost was significantly higher, reaching three orders of magnitude at worst. This is on par with popular dynamic analysis tools such as Valgrind [98]. Our initial implementation of BB-SW in this work is a simple research prototype and is not heavily optimized. Chapter 3 describes a more highly optimized implementation of context-aware communication graph collection.

Graph Sizes

Columns 3-6 of Figure 2.4 show the size of communication graphs with and without context. The size of the communication graph for an execution is determined by the frequency and diversity of inter-thread communication in that execution. The first noticeable trend is that context-aware communication graphs (Columns 5 and 6) are significantly larger than context-oblivious communication graphs (Columns 3 and 4). This is expected, since instructions can execute in multiple contexts. For none of the applications did these graphs exceed 100k nodes and 200k edges in size. Using a straightforward adjacency matrix representation, context-aware graphs never exceed 1 MB in size. The low storage overhead makes it feasible to use context-aware communication graphs during debugging.

2.4.7 Case Study: Configuration error in `dedup`

We conducted a case study using BB-SW to debug a configuration error in `dedup`, one of the PARSEC applications. We reconfigured the hash table data structure used in `dedup` so that it was built with a configuration documented as unsafe for multi-threaded execution — we enabled dynamic hash re-sizing. We then used *BB-SW* to collect graphs from 50 executions, using the buggy configuration. We labeled and processed the collected graphs using our debugging methodology. After examining just 6 code points, we discovered an atomicity violation that occurs when the buggy configuration is used, alongside a developer comment describing that the code was not safe if threading is enabled. While this bug is a documented configuration error, and not a new bug, we consider the ease with which we found the involved code a further validation of the effectiveness of our technique.

2.5 Conclusions, Insights, and Opportunities

Inter-thread communication is a fundamental property of multi-threaded program executions. Understanding differences between communication in failing and non-failing program executions is helpful to understanding concurrency bugs. Communication is helpful because concurrency bugs are the result of unintended interactions between threads and communication is the primary mode of inter-thread interaction.

This chapter described a method for debugging challenging concurrency errors using an abstract graph encoding of inter-thread communication. The key to our approach is in the design of the *context-aware communication graph* abstraction, the representation of communication our technique relies on. Using these graphs we developed Bugaboo, a comprehensive framework for debugging concurrency bugs that manifest as unintended inter-thread communication – including single- and multi-variable ordering and atomicity violations. We developed BB-SW, a software reference implementation of Bugaboo using binary instrumentation and showed that it is effective. BB-SW identifies code points involved in concurrency bugs and did so with run time and space overheads tolerable for debugging.

Insights. There are two important insights in the work in this chapter. First, we discovered that a context-aware communication graph that corresponds to a failing program execution is structurally different from one that corresponds to a non-failing execution. This insight enables using differences in communication graph structure to find concurrency bugs. Looking only at communication, rather than relying on bug-specific heuristics makes our approach to debugging *general*.

Second, we showed that context-oblivious graphs that do not differentiate between different dynamic instances of code points are inadequate for debugging in general. Context-aware communication graphs overcome this limitation by abstractly encoding information about the relative order of communication events in the execution, in the form of *communication context*.

Opportunities. In Bugaboo, we focused on developing, collecting, and characterizing context-aware communication graphs and their role in debugging. The work in Bugaboo creates several interesting opportunities, some of which are covered in future chapters of this dissertation, and some of which are left to future work.

Bugaboo, as discussed in this chapter, does not discuss the role of hardware and architectural support for collecting context-aware communication graphs. Chapter 4 discusses the role of hardware support in implementing Bugaboo.

Bugaboo uses a very rudimentary ranking function to find buggy code points. First, after conversion to the failure-only graph, Bugaboo's ranking function considers only code points, not communication graph edges. Ranking with the information couched in edge

frequencies is a more potent way of discovering code relevant to a bug. Second, Bugaboo reports individual code points related to the bug. Reporting pairs of code points involved in communication, or reporting complexes of code points that reflect communication and temporal sequencing is a better aid to the non-local reasoning required to solve hard concurrency bugs. Chapter 3 addresses these limitations by refining the information that is reported and enriching the ranking function with more information about failed execution behavior.

One major opportunity not addressed by the work in this dissertation is a large scale characterization of communication graphs. A statistical analysis that looks at structural graph properties, or even spectral graph properties may reveal fundamental characteristics of inter-thread communication. Looking at graphs and more general behavioral characteristics (*e.g.*, performance, quality of service) in the large, we may garner insights about how to better write or execute programs. Communication graphs could play a role in IDEs to aid developers. Communication graphs could play a role in dynamic compilers or runtime systems to aid in resource allocation or recompilation decisions. There may even be a role for communication graphs in discovering and enforcing security properties, as these graphs are an abstract representation of information flow among threads.

Chapter 3

RECON: DEBUGGING WITH RECONSTRUCTED EXECUTION FRAGMENTS

Chapter 2 illustrated that Bugaboo is effective for debugging. However, neither Bugaboo, nor other prior approaches (like those discussed in Chapter 8), are a panacea for concurrency bugs. There are several essential limitations to prior techniques. First, many prior approaches to *detect* bugs report *too little* information to understand bugs: a single communication event [81, 121, 139] or the thread preemptions from buggy runs [25, 96]. However, concurrency bugs are complex and involve code points distributed over a code base and in multiple threads, requiring more information to be fully understood. To *understand* such bugs, developers benefit from seeing a portion of the execution illustrating the actual code interleaving that led to a failure. Second, replay-based approaches [137, 106] often report *too much* information – effectively, the entire execution schedule. Replay makes bugs reproducible, but programmers must sift through an entire execution trace to comprehend bugs. Finally, many techniques are tailored to a specific class of concurrency errors [80, 104], limiting their applicability. It is infeasible to anticipate every possible error scenario and design a tool targeting each. Hence, *generality* is crucial.

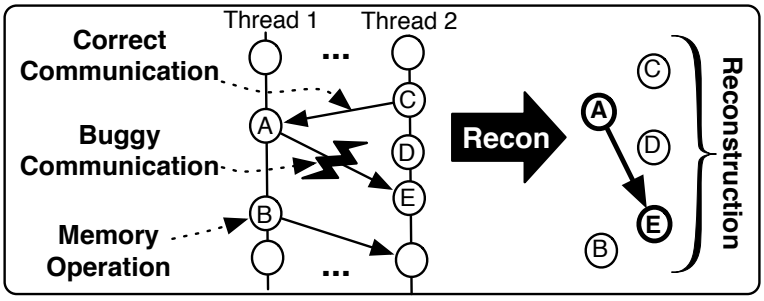


Figure 3.1: **Recon reconstructs fragments of program execution.**

In this chapter, we propose Recon, a new approach to concurrency debugging based on *reconstructions* of buggy executions. A reconstruction is a short, focused fragment of the execution schedule surrounding a shared-memory communication event. Figure 3.1 illustrates what a reconstruction is and how a reconstruction relates to an execution. Reconstructions are based on context-aware communication graphs, as introduced in Chapter 2. Using these graphs, Recon builds reconstructions that show the interleaving that caused buggy behavior, rather than just some of the code points involved. Reconstructions are *general*, as they make few assumptions about the nature of bugs — Recon does not look for bug-specific patterns. Consequently, reconstructions help programmers understand the bugs that lead to failures caused by arbitrary sub-sequences of a multi-threaded execution schedule.

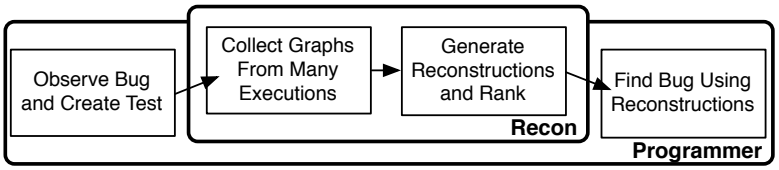


Figure 3.2: **Overview of Recon's operation.**

Figure 3.2 shows an overview of Recon's basic operation. The process begins when a programmer observes a bug or receives a bug report. The programmer then derives a test case designed to trigger the bug, and runs the test multiple times using Recon. Recon collects a communication graph from each execution, and the programmer or test environment labels each graph as buggy or non-buggy, depending on the outcome of the test. Recon then builds reconstructions from edges in buggy graphs; for each one, Recon computes statistical features to quantify the likelihood that it is related to the bug. Recon uses the features to compute a rank for each reconstruction and presents them in rank order.

With Recon, we make several contributions:

- We propose the concept of reconstructing fragments of multi-threaded executions and develop an algorithm that builds reconstructions from communication graphs.
- We propose a set of features to describe reconstructions and use statistical techniques to identify reconstructions that illustrate the root cause of bugs.
- We develop optimization techniques to build communication graphs efficiently.
- We implement Recon for both C/C++ and Java. Our evaluation uses bugs from the literature, including several large applications and shows that Recon precisely identifies bugs and their corresponding reconstructions. We include a case study in which we use Recon to understand and fix an unresolved bug.

3.1 Reconstructed Execution Fragments

The goal of Recon is to simplify debugging by presenting the programmer with a short, focused, temporally ordered reconstruction of the events that were responsible for buggy behavior. Our technique for reconstructing execution fragments is based on a specialized version of the context-aware communication graph abstraction developed in Chapter 2.

Communication Graphs. Before describing how Recon uses context-aware communication graphs, this section briefly reviews the basics of communication graphs and context-aware communication graphs by looking at an example.

Figure 3.3(a) shows an atomicity violation bug that was found in the `mysql` database server. In this example, the two writes to `log_type` in Thread 1 should not be interleaved by reads of `log_type` in another thread. However, a read of the `log_type` variable by Thread 2 may interleave Thread 1's accesses. The result is that database accesses are not logged, which is a security issue. The accesses to `log_type` in this example are not ordered by synchronization, meaning this bug also contains data races. However, even if all accesses were placed in individual critical sections protected by the same lock, the atomicity violation would remain, even in the absence of any data-races.

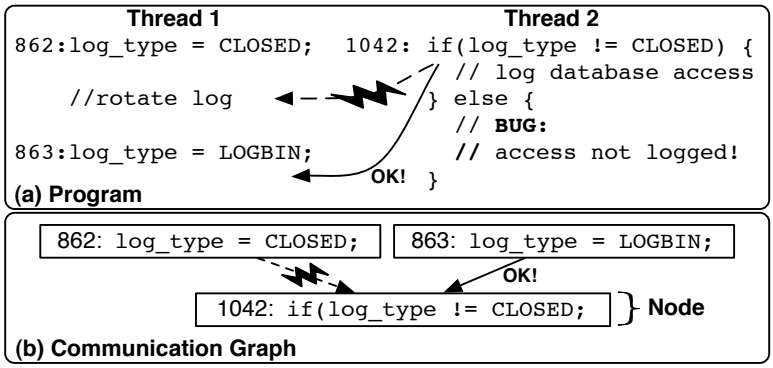


Figure 3.3: **A buggy program and communication graph.** The dashed graph edge represents the buggy communication.

Without guidance, a developer is unlikely to see the following useful, but non-obvious property of the code in Figure 3.3(a): In buggy executions of this program, the read in Thread 2 reads a value written by the first write in Thread 1, but in correct executions the read in Thread 2 reads a value written by the second of Thread 1's writes. Tracking *communication* between instructions in different threads of a program captures this property. Figure 3.3(b) illustrates the communication in buggy and correct executions of this code as a *communication graph*.

A node in a communication graph represents a program instruction. An edge indicates a *communication event* involving the instructions represented by its *source* and *sink* nodes. An edge's source represents the write instruction whose result was read or overwritten by the instruction represented by the sink node.

Using communication graphs, a developer can find the bug in Figure 3.3 by focusing on suspicious communication events that tend to occur in buggy runs, but not in correct runs. Debugging with communication graphs is one of the motivating ideas behind Recon. Rather than just considering the presence or absence of individual communication events in a graph, we reconstruct temporal sequences of communication events and, using machine learning, infer which sequences most likely illustrate a bug's cause.

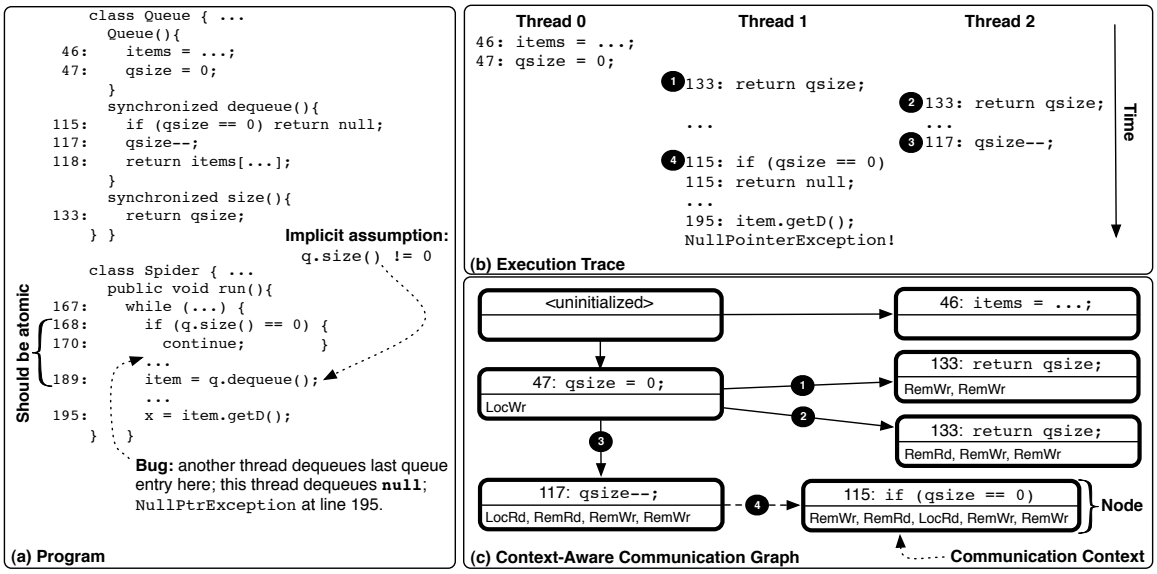


Figure 3.4: **A buggy program, a failing execution schedule, and its context-aware communication graph.** Nodes represent the execution of operations in a specific context. Edges represent communication between nodes. Note that we only include events in nodes' contexts that appear in our abbreviated trace for simplicity's sake.

3.1.1 Context-Aware Communication Graphs

Section 2.1.1 showed that simple, context-oblivious communication graphs are insufficient for general bug detection. Figure 3.4 introduces a running example showing a simplified version of a bug in `weblech`, a multi-threaded web crawler. This example again illustrates the insufficiency of context-oblivious graphs. Figure 3.4(a) shows the program, which has an atomicity violation involving a shared queue. The check of the queue's size on line 168 should be atomic with the dequeue operation performed on line 189 to ensure that there is always an item to dequeue when calling `dequeue()`, but the programmer has not implemented this atomicity constraint. Figure 3.4(b) shows an execution trace manifesting the bug. In this trace, the `size()` and `dequeue()` calls in Thread 2 interleave between the `size()` and `dequeue()` calls in Thread 1. Since the queue is emptied by Thread 2, Thread 1's call to `dequeue()` returns `null`, which is stored in the local variable `item`. Thread 1 later crashes with a `NullPointerException` when trying to invoke the `getD()` method on this null `item`. The problematic communication here is Thread 1's read of the queue's `qsize` field on line

115; it reads a value written by Thread 2 at line 117 rather than the same value Thread 1 read from `qsize` the last time, at line 133. However, looking at this communication alone is insufficient to find the bug, as it occurs in both buggy *and* non-buggy executions.

The solution to this problem is to add *communication context* to nodes. Context is a short (e.g., 5 entries) history of *context events* that occurred before the node's instruction executed. Context events are *Local Reads*, *Local Writes*, *Remote Reads*, and *Remote Writes*. "Local" events are executed by the thread that executed the node's instruction, and "Remote" events are those executed by any other thread. Context events represent memory operations performed by any instruction to any address and only events' types (e.g.,"Remote Read") are stored in a node's context. Context-aware nodes represent the execution of an instruction in a particular communication context, as opposed to just an instruction. Threads watch for events and maintain their *current context*. When a thread communicates, the resulting node is identified by the instruction and the thread's current context.

Figure 3.4(c) shows a context-aware communication graph generated by the trace in Figure 3.4(b). Each numbered circle maps an event in the trace to a corresponding graph edge. Edge 4 occurs when the buggy interleaving occurs, but not in correct executions because its nodes' contexts are unique to buggy executions. Context-aware graphs provide a means to distinguish buggy communication from correct communication in more complex bugs when context-oblivious graphs fail to do so.

Timestamped Communication Graphs

We specialize the context-aware communication graph abstraction to encode ordering between non-communicating nodes, by adding a timestamp to each node indicating when the node's instruction executed. We call this new graph abstraction the *timestamped communication graph*; interchangeably referring to them as just "graphs" hereafter.

Graph Construction. We collect graphs by keeping a *last-writer record* for each memory location composed of: (1) the thread that last wrote the location; (2) the instruction address and context of the write; and (3) a timestamp for the access. When a memory operation

is performed by a different thread than the last-writer, an edge is added to the graph. The edge's source node is populated with the instruction address, context and timestamp stored in the location's last-writer record. The sink node is populated with the instruction address, context, and timestamp of the operation being performed.

To limit the size of the graph, we only record the most recent pair of timestamps for an edge, i.e., the timestamp is not used to identify a node, only the instruction and context are. When adding a communication edge to the graph, if the edge already exists, only the timestamps are updated. By overwriting timestamps, we lose some ordering information, so we call our extension a *lossy timestamp*. Figure 3.5(a) shows an example of a timestamped communication graph. The graph is similar to the one in Figure 3.4, except that each node now has a timestamp indicating when it occurred.

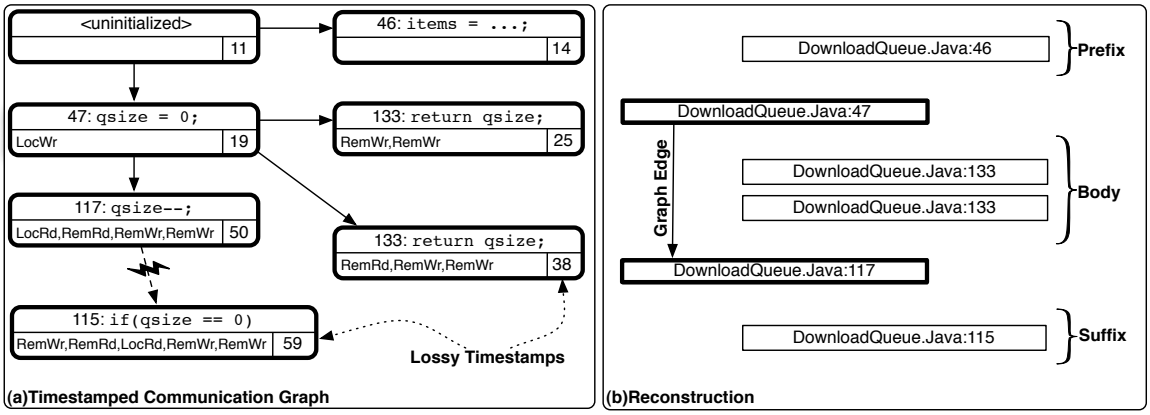


Figure 3.5: **A timestamped communication graph and corresponding reconstruction.** The graph and reconstruction are based on the program, execution schedule, and graph in Figure 3.4.

3.1.2 Reconstructions

A *reconstruction* is a schedule of communicating memory operations that occurred during a short fragment of an execution. In this section, we describe the process of building a reconstruction around a single, arbitrary communication event (i.e., graph edge). Section 3.2 describes how we identify the reconstructions most likely related to bugs, and Section 3.3.3 details the entire debugging process.

Building Reconstructions from Graphs

Recon builds reconstructions starting from an edge in a graph. A reconstruction should include the memory operations that executed in a short window prior to the source node, between the source and the sink nodes, and in a short window following the sink. These *regions* of the execution are called the *prefix*, *body*, and *suffix* of the reconstruction, respectively. Given a graph from a single execution, we can compare nodes' timestamps to determine whether the operation represented by a node occurred in the prefix, body, or suffix of a reconstruction. More precisely, we include nodes timestamped earlier than the edge's source in the prefix. Analogously, we include nodes with timestamps ordered after the edge's sink in the suffix. Finally, the body includes all nodes ordered between the source and the sink.

The size of the window of nodes considered in computing the prefix and suffix is arbitrary. With a larger window, there is a greater chance that unrelated nodes are included in a reconstruction. With a smaller window, fewer unrelated nodes are likely to end up in a reconstruction, but we risk excluding events related to the bug that occur far away from the communication event. A reasonable window size heuristic is to use the length of the communication context of a node. Using the context length, we include nodes that were influenced by, or influenced the context of the sink or source.

Simpler Debugging Using Reconstructions

Figure 3.5(b) shows the reconstruction Recon produces from the graph in Figure 3.5(a). This reconstruction illustrates the buggy interleaving of queue operations shown in Figure 3.4(b). It includes all the code points involved in the bug, and presents them in the order that leads to buggy behavior. In buggy runs, the read of the queue's size on line 133 and the dequeue on line 115 are interleaved by the dequeue at line 117. This buggy interleaving is clear in the reconstruction: line 133's node is in the body and line 115's node is in the suffix. The interleaving dequeue operation at line 117 is the sink node, which is ordered between the body and the suffix. Hence, the reconstruction clearly expresses this buggy interleaving of code.

This example illustrates a key contribution of reconstructions. Looking at the buggy edge between line 117's node and line 115's node does not explicitly indicate the bug – it suggests the involvement of the queue, but not the atomicity violation involving line 133. Instead, the reconstruction built around the *non-buggy* edge between line 47's node and line 117's node illustrates the bug, showing all three involved code points *and* the buggy execution order.

3.1.3 Aggregate Reconstructions

We have described how to build a reconstruction for a single edge from a single execution. We can aggregate reconstructions from a *set* of runs to see how frequently code points occur in a region of a reconstruction *across* executions. This information allows us to define our confidence that a code point belongs in a region.

We compute a reconstruction for each edge in each execution's graph. We then aggregate the reconstructions of each edge across the executions by computing the union of each of their prefixes, the union of each of their bodies, and the union of each of their suffixes, producing the aggregate prefix, body, and suffix. A node may occur in multiple different regions in an aggregate reconstruction, if, for instance, in half of executions it appeared in the prefix, and in half it appeared in the body. Nodes in the same region in an aggregate reconstruction are unordered with one another, but are ordered with the source and the sink of the edge in the reconstruction, and with nodes in other regions. The lack of ordering within a region is because nodes' timestamps may came from different executions, so they cannot be compared.

When aggregating reconstructions, we associate a confidence value with each node in a region. The confidence value is equal to the fraction of executions in which that node appeared in that region. The confidence value of a node in a region represents the likelihood that a node occurs in that region. In Section 3.2, we discuss using confidence values to identify reconstructions likely related to buggy behavior.

Figure 3.6 shows an example of the aggregation process — Figure 3.6(a) shows reconstructions produced from 4 different executions, and Figure 3.6(b) shows the aggregate

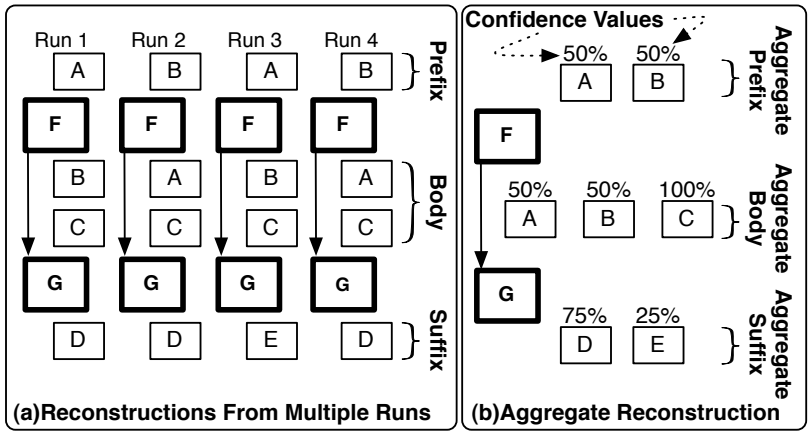


Figure 3.6: **Aggregating reconstructions from many executions.** (a) shows reconstructions of many different failing program executions. (b) shows the resulting aggregate reconstruction with confidence values.

reconstruction produced from these 4 reconstructions. In this example, node A appears in the prefix of half of the reconstructions and appears in the body in half of the reconstructions. The prefix and body of the aggregate reconstruction therefore both include node A, and assign it a confidence value of 50%. Node C appears in the body of all of the individual reconstructions, so it appears in the body of the aggregate reconstruction with a confidence value of 100%.

3.2 Debugging with Reconstructions

Recon's debugging process has four steps. First, the program is run under Recon several times, yielding a set of communication graphs labeled buggy or non-buggy (Section 3.1.1). Second, Recon selects edges from each buggy graph based on how often they co-occur with failure behavior. Third, for each edge selected in the previous step, Recon builds and aggregates reconstructions (Sections 3.1.2 and 3.1.3). The last step is to determine which reconstructions will most likely help a user understand the bug. To do so, we develop *features* describing reconstructions and use them to compute a *rank* for each reconstruction. We now discuss our proposed features.

3.2.1 Features of Reconstructions

A key design concern is that features are *general*. A feature that targets one bug type or pattern is not as useful. If we choose features that are not general, we bias our search toward some bugs and miss others entirely. For example, serializability analysis of memory access interleavings has been used to detect atomicity violations [80, 104]. However, it does not detect ordering bugs, or any multi-variable bugs.

Our features should capture as much information as necessary to discriminate reconstructions of buggy fragments of an execution from reconstructions of non-buggy fragments. The three features we use are: *Buggy Frequency Ratio*, which focuses on the correlation between communication events and buggy behavior; *Context Variation Ratio*, which focuses on variations in communication context that co-occur with failure behavior; and *Reconstruction Consistency*, which looks at the consistency with which sequences of code points occur in reconstructions from failing executions. We describe our features below in detail and verify their efficacy empirically using a feature importance metric from machine learning [68].

Buggy Frequency Ratio (B)

Intuition. A reconstruction's Buggy Frequency Ratio, or B value, describes the correlation between the frequency of the communication event from which the reconstruction was built, and the occurrence of buggy behavior. The motivation for this feature is that we are interested in events in an execution that occur often in buggy program runs, but rarely, or never, in non-buggy runs.

Definition. For each aggregate reconstruction, assume $\#Runs_b$ buggy runs and $\#Runs_n$ non-buggy runs. Assume the reconstruction's edge occurred in $EdgeFreq_b$ buggy runs and in $EdgeFreq_n$ non-buggy runs. The fraction of buggy runs in which the edge occurred is:

$$Frac_b = \frac{EdgeFreq_b}{\#Runs_b}$$

The fraction of non-buggy runs in which the edge occurred is:

$$Frac_n = \frac{EdgeFreq_n}{\#Runs_n}$$

We define B as:

$$B = \frac{Frac_b}{Frac_n} \tag{3.1}$$

If a reconstruction's edge occurs in many buggy runs and few non-buggy runs, B is large. Conversely, if the edge occurs often in non-buggy runs and rarely in buggy runs, B is small.

If the edge never occurs in a non-buggy run, but occurs in buggy runs, then it is very likely related to the bug. However, in such a case, $Frac_n$ is 0. Unless we handle this case specially, B is undefined. In this corner case, we give $Frac_n$ a value that is smaller than the value produced if the edge occurs in one non-buggy run (by assigning $Frac_n = \frac{1}{\#Runs_n+1}$). This yields large B values for these important edges.

Context Variation Ratio (C)

Intuition. The Context Variation Ratio (C) quantifies how variation of contexts of communicating code points correlates with buggy execution. We can determine the pair of communicating code points in the edge around which a reconstruction is built, since a node is identified by an instruction and context. We can then determine all edges involving that pair of code points, regardless of context. After that, we then compute the set of all contexts in which the pair communicated. In a program that has frequent, varied communication, there are many contexts in this set; in a program with little or unvarying communication, the set is small. We consider a reconstruction suspicious if the pair of code points forming the edge around which the reconstruction was built execute in a substantially different number of contexts in failing runs than in non-failing runs.

Definition. For a reconstruction built around an edge between two code points, we define $\#Ctx_b$ to be the number of contexts in which the code points communicated in buggy runs, and $\#Ctx_n$ to be the number in non-buggy runs. C is the ratio of the absolute difference of $\#Ctx_b$ and $\#Ctx_n$ to the total number of contexts for the pair of code points in non-buggy

and buggy runs. We define C as:

$$C = \frac{|\#Ctx_b - \#Ctx_n|}{\#Ctx_b + \#Ctx_n} \tag{3.2}$$

Large C values indicate a disparity in communication behavior between buggy and non-buggy runs. Hence, a reconstruction with a large C value more likely illustrates the communication pattern that led to buggy behavior.

Reconstruction Consistency (R)

Intuition. Reconstruction Consistency (R) is the average confidence value over all code points in an aggregate reconstruction. R is useful because code points that consistently occur in reconstructions of buggy executions are likely related to the cause of the bug. As described in Section 3.1.2, each node in an aggregate reconstruction has an associated confidence value that represents the frequency with which it occurs at a certain point in that reconstruction. In an aggregate reconstruction produced from sets of buggy runs, a node with a high confidence value occurs consistently in the same region of a reconstruction in those buggy runs. Such nodes' operations are therefore likely to be related to the buggy behavior in those runs. Reconstructions containing many high confidence nodes reflect a correlation between the *co-occurrence* of those nodes' code points in the order shown by the reconstruction, and the occurrence of buggy behavior.

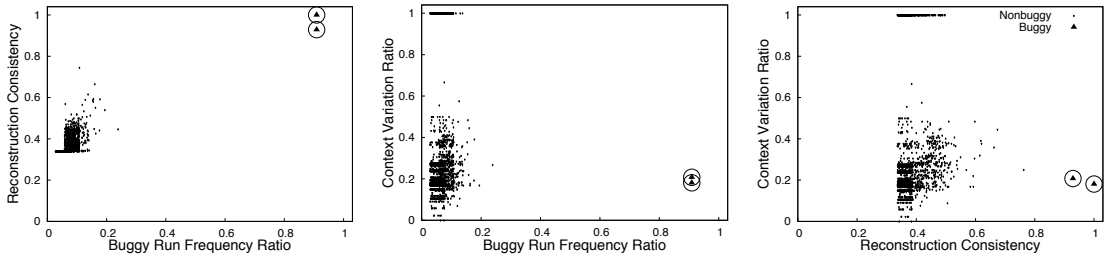
Definition. We compute R for a reconstruction as the average confidence value over all its nodes. Formally, for a reconstruction with prefix region P, body B, and suffix S and where $V(n, r)$ is the confidence value of node n in region r, we define R as:

$$R = \frac{\sum_{p \in P} V(p, P) + \sum_{b \in B} V(b, B) + \sum_{s \in S} V(s, S)}{|P| + |B| + |S|} \tag{3.3}$$

Nodes in a reconstruction with a large R value tend to occur in the reconstructed order when buggy behavior occurs. Such reconstructions are therefore more likely to represent problematic interleavings and to be useful for debugging.

3.2.2 Using Features to Find Bugs

By construction, large values for B, C, or R indicate that a reconstruction is likely to be buggy. Therefore, we give each reconstruction a score equal to the product of all non-zero features' values. We rank reconstructions, with highest scoring reconstruction first.



(a) B vs. R (b) B vs. C (c) R vs. C

Figure 3.7: **Pair-wise feature plots illustrating class separation.** The plots show how effectively each pair of features separate reconstructions of the failure from others for `apache`. We only show the top 2000 ranked reconstructions and points representing reconstructions of the failure are circled.

Empirical validation of B, C, and R. We now quantitatively justify our features using real buggy code (we describe our experimental setup in Section 3.4). We assessed the discriminatory power of our features using Weka's [54] ReliefF [68] feature selection function. The magnitude of a feature's ReliefF value is greater if the distance between points of different classes is greater along that feature's dimension, on average. The magnitude of a feature's ReliefF value corresponds to how well it discriminates between classes.

Table 3.1 shows ReliefF values for several C/C++ applications. All features' ReliefF values are non-zero, meaning they are useful for classification, and many have ReliefF values close to 1.0. The relative importance of features varies by program.

For `apache`, B and R are the most useful. C is less important, indicating there is a similar amount of context variation in buggy and non-buggy runs. Figure 3.7 illustrates the relative importance of the features graphically, with pair-wise feature plots. Figure 3.7(a) shows that when viewed along the axes of highest ReliefF, there is clear segregation of buggy and non-buggy reconstructions. In the plot, buggy points tend to the upper right, meaning

	ReliefF Rank		
Program	B	R	C
apache	0.99	0.91	0.16
mysql	0.20	0.59	0.76
pbzip2	0.26	0.28	0.28
aget	0.84	0.91	0.16

Table 3.1: **Effectiveness of features.** The table shows the reliefF rank of each feature for our C/C++ benchmark programs.

they have larger feature values than non-buggy points. Figures 3.7(b) and (c) reiterate the class segregation along the B and R axes, and illustrate the less clear division along the C axis.

`pbzip2`'s ReliefF values are smaller than other applications' values. The disparity indicates that in each dimension, `pbzip2`'s buggy and non-buggy points tend to be nearer to one another than in other applications. Hence, ranking by a single feature is inadequate to isolate bugs precisely. However, in the three-dimensional space of all features, buggy and non-buggy reconstructions are far apart. As our results in Section 3.4.2 confirm, ranking using all three features isolates the reconstruction of the bug in `pbzip2`.

Our ReliefF feature analysis, emphasizes two properties of our technique: (1) our features precisely classify buggy reconstructions to identify bugs; and (2) considered together, our features are more powerful than each individually.

3.3 Implementation

We implemented two versions of Recon, one for C/C++, using Pin [87], and one for Java, using RoadRunner [46]. The implementation has three parts: (1) tracking communication, (2) collecting graphs, and (3) generating and ranking reconstructions.

3.3.1 Tracking Communication

To track communication, we maintain a *meta-data table*. This table maps each memory location to an entry containing its last-writer record and a list of threads that have read from the location since its last write called the *sharers list*. Each thread has a communication context. A thread's context is a queue of events, as described in Section 3.1. We use a

5-entry context.

When a thread writes to a memory location, it updates the location's last-writer record with its thread ID, the instruction address of the write, its current context, and the current timestamp. If the writing thread is different from the last writer, it does three things: (1) update its context with a local write event; (2) update the context of each thread in the sharers list with a remote write event; and (3) clear the sharers list.

When a thread reads a location, it looks up the last writer thread in the last-writer record. If the reading thread is different from the last writer, it does three things: (1) update its context with a local read; (2) update the last writer thread's context with a remote read; and (3) add the reading thread to the memory location's sharers list.

For C/C++, we implement the meta-data table as a fixed size hash table of 32 million entries. To find a memory location's meta-data, we index with the address modulo the table size. We use a lossy collision resolution policy: on a hash collision, an access may read or overwrite another colliding location's meta-data. We ignore stack accesses, as they are rarely involved in communication. For Java, we use RoadRunner's shadow memory to implement a distributed meta-data table. Its size scales with allocated memory and it does not suffer from collisions. Unique identifiers of memory access instructions in the bytecode replace instruction addresses. Contexts are stored as integers, using bit fields. We instrument accesses to fields and arrays, but not local variables.

3.3.2 Timestamped Communication Graphs

Each thread maintains its own partial communication graph. Partitioning the communication graph over all threads makes adding an edge a thread-local operation, which is critical for performance. When a thread tries to add an edge, it first searches the graph for the edge. If the edge is already in the graph, the thread overwrites the existing timestamps with the timestamps of the edge being added. If not, a new edge is created. When a thread ends, it merges its partial graph into a global graph. Once all partial graphs are merged into the global graph, it is written to a file.

For C/C++, we use the `RDTSC` x86 instruction to track timestamps. Recon maintains

communication graphs as a chaining hash table. Separately for the source and sink node, the hash function sums the entries in each node's context. Each node's sum is then XORed with the node's instruction address. The result of the computation for the source node is then XORed with the result of the computation for the sink, producing the hash key. For Java, we generate timestamps from the system time. We implement communication graphs as adjacency lists, using hash sets. Nodes are indexed by (instruction address, context) pairs.

3.3.3 Generating and Ranking Reconstructions

We generate and rank reconstructions with the following process. We separately load sets of buggy and non-buggy graphs into memory and create a list of nodes ordered by timestamp for each buggy run. At this point, we compute C and B for each edge in the set of buggy graphs. We then rank these edges by their B values. Next, we generate reconstructions for the top 2000 edges ranked by B, using the algorithm described in Section 3.1.2. To limit the size of the reconstructions produced, we limit the number of code points in each region. To do so, we exclude from a reconstruction any node that has a confidence value less than half the region's maximum confidence value. After computing reconstructions, we compute their R values and their ranks, and output them in rank order.

3.3.4 Optimizing Graph Collection

We use two optimizations to reduce overheads: (1) we reduce the number of instructions for which analysis is required and (2) we permit an instrumentation data-race to avoid locking overheads.

Selectively Tracking Memory Operations

The simplest way of reducing graph collection overhead is monitoring fewer memory operations. We develop two optimizations to do so. They can lead to lost or spurious edges, but our results (Section 3.4) show that Recon's accuracy is unaffected.

First Read Only. Repeated reads to a memory location by the same thread are likely redundant. We therefore develop the *first-read* optimization: threads only perform analysis on their first read to each location after a remote write to that location. Not performing updates on these subsequent reads is analogous to performing analysis only on cache read misses. Due to the frequency and temporal locality of reads, this optimization eliminates many updates.

This optimization is *lossy*. If a thread repeatedly reads the result of a write, only its first read is reported. If subsequent reads are performed in different code with different contexts they will not cause edges to be added to the graph. Additionally, context events corresponding to ignored reads are not published to threads' contexts, which may result in fewer distinct contexts and edges.

First Write Only. Repeated writes by the same thread are often redundant or non-communicating. Under the *first-write* optimization, a thread only updates the last-writer table and sharers list on a write to a memory location x when it is not the last thread to write x. This optimization is *noisy*. If a thread that is not the last writer of x writes to x and does not update x's meta-data on subsequent writes, another thread's read of x may see outdated meta-data and add a spurious edge with incorrect context information to the graph.

Intentional Instrumentation Races

On every memory access, threads check the last writer of the location they are accessing to determine what analysis operations must be performed, as described in Section 3.3.1. To ensure threads observe consistent meta-data, they acquire a lock on each access.

We observe, however, that due to temporal locality these checks are often performed by the location's last writer. In such situations, reading all meta-data is unnecessary. The cost of acquiring the lock just to check the last writer outweighs the cost of the check itself. To mitigate this cost, we can perform the check without holding the lock, which we call the *racy-lookups* optimization. If, based on the check, a thread determines it must perform further analysis or update the meta-data, it acquires the lock. Only the check to determine

the location's last writer races.

In principle, data-races can lead to undefined behavior [24] or memory inconsistency [88]. In practice, there are only two inconsistent outcomes of this optimization. The first is that the last writer performs a check that indicates it is not the last writer. In this case, the checking thread last wrote the meta-data. On x86 the thread will correctly read its own write, making this situation impossible. In Java, our meta-data writes are well ordered and the read involved in the check is ordered with its meta-data update. As a result, the checking thread can only ever correctly read that it was the last writer. The other inconsistency is when a check indicates to a thread that it is the last writer when it is not. This situation is possible in x86 and Java. On x86, the check reports that the checking thread is the last writer, so it does no analysis. Because the check was not synchronized, however, another thread's update to the last-writer field may have been performed, but not yet made visible to all threads. In this case, the checking thread should have seen the update and added an edge, but it did not. This situation can also arise because our instrumentation is not atomic with program accesses.

In practice, this effect has little impact on our analysis. Furthermore, the statistical nature of Recon is robust to noise, so such omissions do not impact Recon's bug detection capability.

3.4 Evaluation

There are several components to our evaluation. We show that our ranking technique is effective at finding bugs and the reconstructions Recon produces are useful and precise. We show that Recon remains effective with very few program runs. We describe a case study of our experience fixing a previously unresolved bug. Finally, we show that with our optimizations, Recon's overheads are similar to the overheads of other analysis tools and overall data collection time is short.

3.4.1 Experimental Setup

We evaluated Recon's ability to detect concurrency bugs using the buggy programs described in Table 3.2. We used a set of full applications, as well as several bug kernels. Our

	Category	Program	Version	Bug Type
C/C++	Bug Kernel	logandswp	n/a	Atomicity Violation
		circlist	n/a	Atomicity Violation
		textreflow	Mozilla 0.9	Multi-Variable Atomicity Violation
		jsstrlen	Mozilla 0.9	Multi-Variable Atomicity Violation
	Full App.	apache	httpd 2.0.48	Atomicity Violation
		mysql	mysqld 4.0.12	Atomicity Violation
		pbzip2	pbzip2 0.9.1	Ordering Violation
		aget	aget 0.4	Multi-Variable Atomicity Violation
Java	Bug Kernel	stringbuffer	JDK 1.6	Multi-Variable Atomicity Violation
		vector	JDK 1.4	Multi-Variable Atomicity Violation
	Full App.	weblech	weblech 0.0.3	Atomicity Violation

Table 3.2: **Buggy programs used to evaluate Recon.** We used both C/C++ programs and Java programs and we included a variety of bug types.

bug kernels are shorter programs with bugs extracted from the literature (`stringbuffer`, `vector`, `circlist`, `logandswp`), and buggy sections of code extracted from the Mozilla project (`textreflow`, `jsstrlen`). Our benchmarks encompass many bug types observed in the wild [79] including ordering bugs and single and multiple variable atomicity bugs. We ran each application in Recon with all optimizations. Our test script labeled graphs based only on externally observable failure symptoms (e.g., crashes, corrupt output, etc.).

We evaluated Recon's runtime and memory overhead using the PARSEC benchmark suite [17] with its `simlarge` input for our C/C++ implementation. For Java, we used 6 applications from the DaCapo benchmark suite [18] with default inputs and all the Java Grande benchmarks [122] with size A inputs. We ran PARSEC and Java Grande with 8 threads; we let the DaCapo benchmarks self-configure based on the number of processors and did not instrument the DaCapo harness. We also ran 4 additional full applications, each with 8 threads: `mysql`, a database server, tested using the sysbench OLTP benchmark with the default table size (10,000) for the performance measurements and table size 100 for debugging; `apache`, a web server, tested using ApacheBench; `aget`, a download accelerator, tested fetching a large web file; and `pbzip2`, a compression tool, tested compressing a 100MB text file. For performance measurements, we ran the uninstrumented version and Recon, with the *first-read*, *first-write*, and *racy-lookups* optimizations. We also ran three less-optimized configurations to understand the impact of each optimization: "Base" analyzes all memory accesses; "FR" uses just the *first-read* optimization; "FR/W" adds the *first-*

Program	Rank of Bug	# Code Pts. Rel.	# Code Pts. Irr.	In Order?	Code Pts Missing	Sensitivity To # Buggy w/ 5	Sensitivity To # Buggy w/ 15	Collect Time (h:m:s)
logandswp	1	6	1	Yes	0	1	1	—
circlist	1	3	3	Yes	0	1	1	—
textreflow	1	8	0	Yes	0	1	1	—
jsstrlen	1	7	0	Yes	0	1	1	—
apache	1	5	5	Yes	0	1	1	0:27:32
mysql	1	8	7	Yes	0	34	9	0:07:08
pbzip2	1	11	0	Yes	1	2	1	1:51:56
aget	1	4	2	Yes	0	8	1	0:59:41
stringbuffer	1	6	0	Yes	0	1	1	—
vector	1	6	0	Yes	0	1	1	—
weblech	1	6	28	Yes	0	4	1	0:13:36

Table 3.3: **Properties of reconstructions for our benchmarks.**

write optimization. We ran all experiments on an 8-core 2.8GHz Intel Xeon with 16GB of memory and Linux 2.6.24. The Java tool used the OpenJDK 64-bit Server VM 1.6.0 with a 16GB max heap. We report results averaged over 10 runs of each experiment.

3.4.2 How Effectively Does Recon Find Bugs?

We produced reconstructions from the graphs we collected, and ranked them as described in Section 3.2.2. We examined the highest-ranked reconstruction that illustrated the bug and analyzed the key properties of that reconstruction. Table 3.3 summarizes our findings.

False Positives. The most important result in Table 3.3 is that for all applications, *the top-ranked reconstruction revealed the bug*, as shown in Column 2. This result demonstrates that our ranking technique effectively directs programmer attention to buggy code with no distracting false positives. This result also corroborates the results from Section 3.2.2, showing that our features precisely isolate buggy reconstructions.

Unrelated Code in Reconstructions. Columns 3 and 4 in Table 3.3 show the number of relevant and irrelevant code points were included in the bug's reconstruction. We consider a code point related if it performed a memory access that read or wrote a corrupt or inconsistent value, or if it is control- or data-dependent on the buggy code. In most cases, virtually all code in the reconstruction is relevant to the bug. However, some reconstructions include code points unrelated to their bug. In `aget`, the two irrelevant code points are in

straight-line code sequences with related code points, at a distance of less than five lines. Such nearby but irrelevant code is not likely to confuse a programmer.

In `mysql`'s case, five out of seven unrelated code points are in straight-line sequences with relevant code. The remaining two in `mysql`'s reconstruction, and all five in `apache`'s reconstruction, were not in straight-line code with relevant points. Instead, they were in another function that was the caller or a callee of a function containing relevant code. Developers debugging programs are likely to understand such caller-callee relationships, suggesting that these code points will not be too problematic.

`weblech` had several irrelevant code points in its reconstruction (28). The reason for their inclusion is that the bug usually occurs at the start of the execution. At this point, constructors have only just initialized data at a variety of code points in the program, resulting in many edges being added between initialization code and other code. The initialization code is easy to identify, especially with program knowledge. These code points clutter the reconstruction, but the bug is reported accurately.

Reconstruction Order Accuracy. Column 5 shows whether or not the code points in the reconstruction were shown in the order leading to buggy behavior. Code points appear in an order that leads to buggy behavior in all cases. In `logandswp`, the last code point in the buggy execution order appears in both the prefix and suffix of the reconstruction, because the code point is in a loop, however, the buggy interleaving is clear.

Missing Code Points. Column 6 shows the number of code points directly involved in the bug that were omitted from the reconstruction. Only one case lacked any involved code points: The code points in `pbzip2`'s reconstruction all relate to establishing the corrupted state condition required for a crash to occur. The actual crashing access is not included.

Sensitivity to Number of Buggy Runs. Columns 7 and 8 illustrate Recon's sensitivity to the number of buggy runs used. Column 7 shows the rank of the bug's reconstruction using 25 non-buggy runs and 5 buggy runs. Column 8 shows the rank using 25 non-buggy and 15 buggy runs. Even with very few buggy runs, Recon gives a high rank to

reconstructions of the bug. Using fewer buggy runs does not impact precision substantially, except for `mysql`. Excluding `mysql`, Recon ranked the bug's reconstruction 8^{th} or better with just 5 buggy runs, and first with 15 buggy runs. For `mysql`, using fewer runs caused Recon to rank some non-buggy reconstructions above the bug's — 33 with 5 buggy runs, and 8 with 15 buggy runs. As shown in Column 2, Recon *always* ranked the bug's reconstruction first with 25 buggy runs. These results show that with very few buggy runs, Recon can find bugs with high precision. In cases where a small number of buggy runs is insufficient, adding more runs increases Recon's precision.

Graph Collection Time. Column 9 shows that the total time required for graph collection (for 25 buggy and 25 non-buggy graphs) is small. In our experiments, all applications took under two hours; `apache`, `mysql`, and `weblech` all took under 30 minutes. These data show that Recon is not only effective at detecting real bugs in these full applications, but also reasonably fast.

3.4.3 Case Study: Debugging an Unresolved Bug

The `weblech` bug is open and unresolved in the program's bug repository. While the bug has been discussed previously [63], we were unaware of any details of the bug prior to this case study. We used Recon to find the problem, and we were able to write a fix using Recon's output and our limited program knowledge.

We began with a bug report describing intermittent non-termination. Using the input from the report, we were able to reproduce the bug in about 1 in 15 runs. We then ran the application repeatedly and watched the output to identify the hang. We noticed that, consistently, at least one thread crashed on a null pointer dereference during hanging runs. We collected 25 buggy and 25 non-buggy runs, identifying bugginess by watching for unhandled exceptions. We then produced reconstructions from these runs.

The first reconstruction reported was related mostly to object constructors, but also included evidence of several accesses to a shared queue data structure, as well as a suspicious `while` loop termination condition involving the queue's size. The body of the reconstruction contained the initialization of and accesses to the size of the queue. The sink of the recon-

	Name	Slowdown (x) Recon	FR/W	FR	Base		Name	Slowdown (x) Recon	FR/W	FR	Base
Apps.	weblech	1.1	1.2	1.2	1.1	PARSEC	dedup	5.5	5.8	5.8	13.8
	pbzip2	1.3	1.3	1.3	1.5		canneal	6.8	6.8	6.5	14.9
	aget	1.9	1.9	1.9	1.9		freqmine	8.8	52.6	56.6	223.8
	apache	5.4	31.7	31.7	177.1		fluidanimate	9.8	9.9	10.1	9.8
	mysql	23.9	102.1	127.2	129.9		streamcluster	10.1	10.1	10.3	10.1
DaCapo	pmd	5.6	5.8	6.0	6.3		blackscholes	14.4	17.8	18.0	40.9
	avrora	5.6	7.9	10.3	27.8		ferret	14.6	70.9	73.1	537.3
	tomcat	6.9	6.2	6.7	9.1		bodytrack	14.9	116.2	120.8	595.5
	xalan	7.1	7.0	7.5	10.8		facesim	15.8	18.8	19.2	29.2
	luindex	8.2	9.1	14.3	20.6		swaption	17.9	96.0	100.6	383.7
	lusearch	17.3	18.1	22.7	22.6		x264	18.9	218.4	236.8	697.4
Java Grande		74.9	85.1	88.4	563.7		vips	28.8	230.6	257.6	996.8

Table 3.4: **Performance of Recon.** We shows Recon's base configuration and many less-optimized configurations relative to uninstrumented execution.

struction's edge was an access to the queue data structure in the dequeue method. In the suffix of the reconstruction was another call to the queue's dequeue method. As we described in Section 3.1.1, such an interleaving of a dequeue call between an access to the queue's size and a subsequent dequeue call violates the atomicity of the pair of operations. The atomicity violation leads to a thread crashing early due to the `NullPointerException` we observed. Crashing prevents the thread from correctly updating the variable for the `while` loop to read. The crash is therefore also responsible for the program's non-termination, as described in the bug report. We fixed the bug by extending a `synchronized` block including the queue size check and the dequeue. With our fix, we didn't see the buggy behavior in several hundred runs – we conclude that we fixed the bug based on the information provided by Recon.

3.4.4 Performance

In Table 3.4, we report run times relative to uninstrumented execution for Recon and the three less-optimized configurations. The main result is that Recon imposes slowdowns as low as 34% for C/C++ (`pbzip2`) and 13% for Java (`weblech`).

Slowdown for full applications never exceeds 24x, even during an industrial strength test of a commercial database (`mysql`). For PARSEC, we saw slowdowns ranging from 5.5x to 28x, showing that Recon performs well on applications with a variety of sharing patterns.

We saw comparable results for DaCapo: overheads of Recon ranged from 5.6x to 17.3x.

Interestingly, overheads tended to be more severe for applications that perform *infrequent sharing*, than those that share often. For example, `dedup`, which uses shared queues, and `avrora`, which exhibits a high-degree of fine-grained sharing [18], both had fairly low overheads, around 6x. In contrast, `swaptions` has infrequent synchronization [17] and threads in `lusearch` interact very little [18] – both suffered higher overheads, around 18x. This trend is further illuminated by the Java Grande benchmarks; these are primarily data-parallel scientific computations that perform little sharing [122]; their average overhead is 75x. Nonetheless, Recon is efficient in applications with high-frequency sharing and for all the mainstream applications we tested.

Effectiveness of Optimizations. Comparing "FR" with "Base" and "Recon" with "FR/W" in Table 3.4, we see that the first-read and racy-lookup optimizations, respectively, significantly improve performance. Comparing "FR/W" and "FR", we see that the first-write optimization has a less significant effect in general — likely because writes are less common than reads — but for `mysql` and `lusearch`, the first-write optimization is clearly important.

The data show that our optimizations are essential to Recon's efficiency. For many applications, our optimizations reduce Recon's slowdown by orders of magnitude. `apache` is one such application: without optimizations, `apache`'s slowdown is 177x, making full-scale tests nearly impossible due to timeouts and unhandled delay conditions in the code. Optimizations reduce this to just 5.4x, enabling Recon to be used with real bug-triggering inputs.

In our experiments, we used PARSEC's `simlarge` inputs to make experimenting with unoptimized configurations feasible, but there is no need to scale inputs for use with Recon. We also experimented with PARSEC's `native` input, using Recon with all optimizations: Experiments finished quickly, and we saw slowdowns nearly identical to the `simlarge` input.

The optimizations have less impact on our Java implementation, but still account for significant speedups (e.g., `avrora`, `luindex`). For most Java benchmarks, the racy-lookup optimization had little effect. Java uses several techniques that significantly reduce the cost

of acquiring locks [67]. It is likely that the racy-lookup optimization is less beneficial than in C/C++ because the cost of locking is lower in Java to begin with.

Memory Overhead. The C/C++ Recon implementation uses a fixed-size 4GB meta-data table, dominating memory overheads in our experiments. Graphs are small in comparison. The table is large enough that the impact of hash collisions was negligible. In a memory-constrained setting, a smaller table could be used at the expense of decreased precision due to hash collisions. In Java, each field and array element is shadowed by a meta-data location: memory overhead scales roughly linearly with the program's footprint. Peak overheads in the optimized version ranged from 2.5x to 16x.

3.5 Conclusions, Insights, and Opportunities

In this chapter we introduced Recon, a novel and *general* approach to isolating and understanding all types of concurrency bugs. Recon works by reconstructing fragments of buggy executions that are likely the result of a bug, providing sufficient yet succinct information to help programmers understand the cause of concurrency bugs, rather than just showing the code involved or reproducing an entire buggy execution.

Reconstructions show the schedule of execution that led to the bug, clearly exposing its root cause. Reconstructions are built by observing multiple executions of a program and collecting timestamped context-aware communication graphs, which encode information about the ordering of inter-thread communication events. We developed a simple statistics-based approach to identify buggy reconstructions. We proposed three bug-independent features of reconstructions that together precisely isolate reconstructions of buggy executions. In order to provide efficient collection of timestamped graphs, we used several techniques that significantly reduce runtime overheads. We implemented Recon for C/C++ and Java and evaluated it using large software. Our results show Recon reconstructs buggy executions with virtually no false positives, and that collecting the data comprising reconstructions takes just minutes.

Insights. There are several key insights in this chapter. We showed that information about communicating instructions is only part of the concurrency debugging story.

Temporal sequencing information together with communication better illustrates bugs than either on their own. Reconstructed execution fragments from failing program executions illustrated that failure behavior is *consistent*. The important parts of an execution happen reliably when the failure happens. Reporting those parts in reconstructions is what makes Recon effective.

Opportunities. Recon paves the way for future research in several directions. Reconstructions do not include information about the data involved in the operations that make them up. Including information about which data structures or which variables are involved may be a helpful way of providing more information to programmers.

Recon's overheads, while substantially lower than many similar techniques, are still high in some cases. It would be interesting in the future to study the trade-offs of the information a system collects, for the debugging value of that information. Sampling instrumentation is one possibility for reducing overheads. Static control-flow and data-flow analyses may also yield insights into instrumentation points that are redundant. Eliminating redundant instrumentation would reduce overheads. As Chapter 4 will describe, hardware support can accelerate communication graph collection. Integrating a hardware design with Recon's debugging methodology is also likely to be profitable.

Other work that is likely to be of interest in the future is to use Recon's approach to address nondeterministic performance problems in concurrent code. If the performance issues are due to concurrent behavior that manifests in communication graphs, reconstructions may be helpful for programmers trying to understand their cause. A key challenge is that while correctness, as framed in this work, is a binary criterion – the program fails or does not fail – performance is continuous, for example, as throughput varies between 0 and an observed maximum. It is likely that performance is modal; different reconstructions may correspond to different performance modes, helping understand the cause of the modality.

Chapter 4

ARCHITECTURE SUPPORT FOR CONTEXT-AWARE COMMUNICATION GRAPHS

Chapters 2 and 3 described Bugaboo and Recon, two techniques for debugging concurrency errors based on context-aware communication graphs. One of the main sticking points in Bugaboo and Recon is the high run time overhead of the software-based tools used to collect graphs. Recon's optimizations are designed to make the time requirements manageable, but with a 75X worst case slowdown, collecting 25 to 50 communication graphs can take a very long time in some cases.

Bugaboo's and Recon's runtime overheads are typically practical for debugging; the alternative is to spend the weeks or months (or even *years* in some cases [138]) to fix concurrency bugs using conventional techniques. However, many concurrency bugs manifest only rarely, requiring many program executions. Furthermore, collecting graphs only "in the lab", during debugging ignores a wealth of data available "in the wild". Production systems are executing widely deployed programs effectively constantly. Collecting graphs from every deployed system executing a program – when crashes occur and during non-failing execution – would alleviate the pain of graph collection during debugging.

The key to *in situ* graph collection is a mechanism for collecting graphs that imposes a run time overhead on the execution that is tolerable for *end users*. The overhead tolerability threshold for end users is substantially lower than for developers. In this work, we develop an architecture-level implementation of a context-aware communication graph collection mechanism that aims for *zero* run time overhead. Such a mechanism requires the cost and complexity of custom hardware, but after paying that tax up front, users of systems can collect context-aware communication graphs unfettered by software slowdowns.

4.1 CACG-HW: Architectural Support for Context-Aware Communication Graphs

Implementing context-aware communication graph collection using architecture support requires a mechanism for monitoring communication between processors. When communication occurs, such an implementation can update the graph by adding edges and can update threads' context queues. This section assumes a graph formalism that is the same as the one discussed in chapters 2 and 3 and describes CACG-HW, our proposed architecture support for **C**ontext-**A**ware **C**ommunication **G**raph collection in **H**ard**W**are.

CACG-HW tracks communication graph edges and context events as cores in a multiprocessor communicate. The cache coherence protocol in typical shared-memory multiprocessor systems provides much of the support that CACG-HW needs to track communication. Whenever threads executing on different processors communicate, coherence messages are sent between those processors. CACG-HW takes advantage of existing coherence message support and a small amount of cache line meta-data to efficiently implement the last writer table in hardware. In our design, each processor's cache records the instruction address and context of the last write to each of its lines. When communication occurs and a coherence message is sent between processors, the coherence message includes the last writer meta-data. When a processor receives a cache coherence message containing meta data, it has the last writer information that it needs to construct a new communication graph edge.

CACG-HW has five components: (1) a per-processor context register that keeps track of recent communication events; (2) coherence message extensions to carry last writer meta-data information; (3) cache line extensions to track the instruction address and context of each line's last write; (4) a software-visible table to store communication graph edges as they are added to the graph; and (5) a software runtime that periodically reads the graph edges collected, preserving the graph for debugging. Figure 4.1 shows an overview of CACG-HW's extensions to a commodity multiprocessor.

Tracking the Communication Context

CACG-HW treats cache-to-cache transfers as potentially communicating memory operations that generate context events. The four context events discussed in Section 2.1.2 map

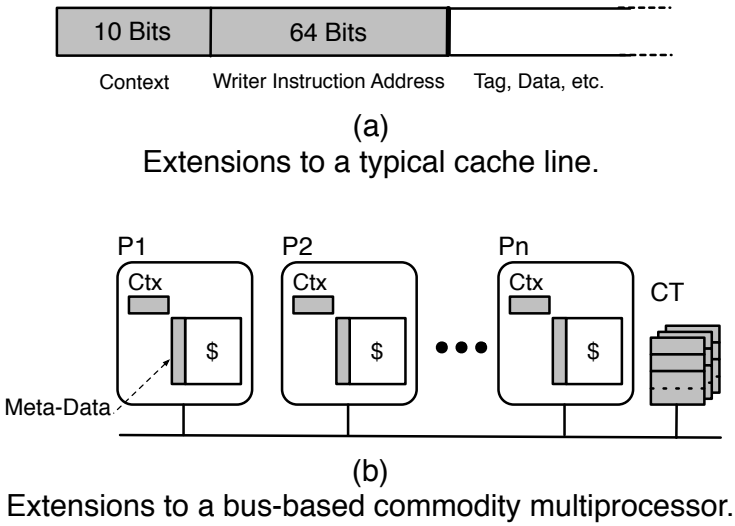


Figure 4.1: **CACG-HW architectural extensions to a typical multiprocessor.** New architectural components are shaded.

directly to cache coherence events. We assign each relevant cache coherence event a two-bit code: local read miss (LcRd); local write miss or upgrade miss (LcWr); incoming invalidation (RmWr); and incoming read request (RmRd).

To implement the context queue, we use a FIFO of context events. The FIFO is used as a shift register. When a coherence event occurs, its corresponding event code is shifted into the context queue FIFO. Each processor in the system has its own context register (Figure 4.1(b)). As in BB-SW, we keep five events of context, so our context registers are ten bits long.

Last Writer Meta-Data and Coherence Message Extensions

Precisely keeping track of communicating instructions requires keeping last-writer information at the granularity of words (or potentially bytes) for all of memory. This is too complex and expensive to implement in hardware. We choose two simplifications to reduce hardware complexity at the cost of some information loss. First, we track communication at cache-line granularity. False sharing might lead to edges that are not actual communication. Second, we monitor only inter-thread communication that happens via cache-to-cache

transfers. When a line with last writer meta-data is evicted, its meta-data is discarded. Future accesses to the evicted line will not record communication involving the operation in the discarded meta-data. As our data show (Section 4.2), the information loss is not significant and does not limit the bug detection capability of our techniques.

We add meta-data to each cache line to keep track of the instruction address and context of the last write to the line, which together are the source node of a graph edge. The meta-data stores the last writer instruction's virtual address and the contents of the last writer processor's context queue at the time when the last writer instruction was executed. Assuming a 5 entry context, comprising 10 bits and an instruction address width of 64 bits, the total space overhead is 74 bits per cache line. Using a narrower instruction encoding (*e.g.*, 48 bits) or a communication context with fewer events would result in lower space overhead. Figure 4.1(a) illustrates our extensions to the cache line.

A cache line's last writer meta-data is updated when a processor writes to a line that is in neither exclusive nor modified state, i.e., during a write or upgrade miss. As a result, meta-data updates are only as frequent as write/upgrade misses.

Last writer meta-data is carried between processors in an extension to existing coherence messages. Without loss of generality, we assume the underlying system has a MESI cache coherence protocol. We augment *read reply* and *invalidate acknowledgment* coherence messages to include meta-data extensions.

- **Read Reply** When a processor receives a read request for a cache line that it has cached, it sends a read reply to the requester. The processor includes last writer meta-data for the involved line in its read reply coherence message. This coherence extension enables CACG-HW to track read-after-write (RAW) communication.

- **Invalidate reply (ack)**. When a processor receives an invalidation message for a line that it has cached, it sends an invalidation acknowledgment to the processor that sent the invalidation. The recipient of the invalidation includes the last writer meta-data of the invalidated cache line in its invalidation acknowledgment coherence message. This coherence extension enables CACG-HW to track write-after-write (WAW) communication.

Communication Table

The purpose of the *Communication Table (CT)* is to store communication graph edges during execution. The CT is organized as a fixed size FIFO queue. Each entry contains the instruction address and context of the *source* and *sink* of a communication edge. The size of each entry is 148 bits – 64 bits for each of the node's instruction addresses, summing to 128 bits and 10 bits for each node's context, together contributing the remaining 20 bits. The CT, shown in Figure 4.1(b), can be organized either as a centralized or distributed data-structure. Since there are no global consistency properties that need to be kept, each processor can have its own CT, posing no scalability issues.

There are three events that require an edge to be added to the graph, leading to a write to the CT: invalidate acknowledgment events, read reply events, and read misses serviced from memory.

- **Invalidate acknowledgment**. When a processor attempts to write a line and sends an invalidation to other processors, it collects invalidate acknowledgment coherence messages. Invalidate acknowledgments in CACG-HW contain meta-data describing the last writer of the line being invalidated. When the invalidating processor receives an acknowledgment containing meta-data, it adds an entry to the CT for a new communication graph edge. The source of the edge is created using the meta-data in the acknowledgment. The sink of the edge is created using the address of the instruction that caused the invalidation to be sent and the contents of the invalidating processor's context queue. Adding a CT entry on receiving an invalidation acknowledgment captures inter-thread WAW communication.

- **Read reply**. When a processor attempts to read a line and sends a read request to other processors, it collects read reply coherence messages. Read replies in CACG-HW contain meta-data describing the last writer of the line being requested. When the requesting processor receives a read reply containing meta-data, it adds an entry to the CT for a new communication graph edge. The source of the edge is created using the meta-data in the read reply. The sink of the edge is created using the address

of the instruction that caused the read request and the contents of the requesting processor's context queue. Adding a CT entry on receiving a read reply captures inter-thread RAW communication.

- **Read miss serviced from memory**. When a processor sends a read request for a line and receives no replies containing meta-data from other processors, the read miss is serviced from memory. When the requesting processor determines a read request is serviced from memory, it adds an entry to the CT for a new communication graph edge. In this case, the edge represents an access to a line that is uninitialized or has not been recently shared. The source of the edge is created with a special "NULL" instruction and context. The sink of the edge is created using the address of the reading instruction and the contents of the reading processor's context queue. Adding a CT entry for read misses serviced from memory captures accesses to uninitialized or infrequently shared data.

4.1.1 Context-Aware Communication Graphs in Production Systems Using CACG-HW

CACG-HW's support for collecting context-aware communication graphs is off the critical path of an execution, so it causes negligible performance degradation. Therefore we can use it in a deployment scenario to continuously collect an execution's communication graph. When a failure occurs during a production execution on a deployed system, the graph produced from that failing execution can be collected and sent back to developers for debugging using the techniques described in Bugaboo. Periodically, graphs from non-failing executions could also be sent back to developers for use in program understanding or debugging.

4.2 Evaluation

We evaluated our design of CACG-HW to characterize its overheads and justify our design choices.

4.2.1 Experimental Setup and Methodology

We evaluated CACG-HW using a simulator based on Pin [87] and SESC [115]. The simulator models a 16-node multiprocessor, with 32KB 8-way associative L1 Caches – a design that is similar to the Intel Core architecture. We modeled a MESI cache coherence protocol enforcing coherence at the L1. We modeled our cache meta-data extensions, communication context queues, coherence message extensions to transfer meta-data, and a 16k-entry communication table that generated traps to our software layer. We used the same benchmarks to evaluate CACG-HW that were discussed in Section 2.4.1.

Benchmark	**# of Inspections Required**
BankAcct	**1.4** (1.0)
CircularList	**2.2** (1.2)
LogAndSweep	**1.0** (1.0)
MultiOrder	**1.0** (1.0)
Moz-jsStr	**1.8** (1.0)
Moz-jsInterp	**1.0** (1.0)
Moz-macNetIO	**1.4** (1.0)
Moz-TxtFrame	**1.0** (1.0)
MySQL-IdInit	**1.0** (1.0)
MySQL-BinLog	**34.0** (24.4)
Apache-LogSz	**12.0** (10.6)
PBZip2-Order	**14.5** (4.8)
AGet-MultVar	**1.0** (1.0)

Table 4.1: **Bug detection accuracy using CACG-HW.** We report the number of code point inspections required before the corresponding bug was found. The number in parenthesis show the number of distinct functions. Note that one inspection indicates that zero irrelevant code points needed inspection, since the bug was found on the first. Results are averaged over five trials.

4.2.2 Debugging Efficacy

We collected communication graphs using our simulated hardware implementation of CACG-HW for the applications listed in Column 1 of Table 4.1. We applied the debugging methodology described in Chapter 2 and in Column 2 of Table 4.1, we report the number of code inspections required to isolate a code point related to the bug in each benchmark. The key result is that in few code point inspections (between 1 and 34 points) or function inspections (between 1 and 24) a programmer is led to code involved in the bug. The results show that CACG-HW yields results very similar to those obtained using BB-SW, the software

graph collection tool described in Chapter 2. This data illustrates that with hardware support, little enough precision is lost in graph collection that our debugging technique remains effective.

4.2.3 Impact of Graph Collection Imprecision

As discussed in Section 4.1, we traded precision for lower complexity in CACG-HW. Compared to whole-memory communication tracking at word granularity, there are two sources of imprecision in CACG-HW: (1) communication tracking at a cache-line granularity; and (2) considering only cache-to-cache transfers as communication (*i.e.*, the impact of evictions). We quantify imprecision introduced by line-level tracking and evictions by computing the number of distinct communicating code points that were present in the graph collected using whole-memory word-level tracking (as used by BB-SW) but *not* present in the graph produced by CACG-HW. Table 4.2 breaks down the results by the source of imprecision. Column 2 shows the impact of line-granular tracking and Column 3 shows the impact of line-granular tracking, combined with the impact of evictions. Column 2 shows that imprecision added by line tracking ranged from 17% to 27% and Column 3 shows that imprecision added by cache-to-cache and line tracking is about twice that.

Benchmark	**% Imprecision Introduced**	
	Line Granularity	**Line Gran. & Evictions**
MySQL-BinLog	27.43	54.51
Apache-LogSz	25.57	59.09
PBZip2-Order	26.32	59.65
AGet-MultVar	16.67	33.33

Table 4.2: **Imprecision for different configurations of CACG-HW.** We show imprecision for line-level and cache-to-cache-only tracking of inter-thread communication.

Section 4.2.2 showed that neither source of imprecision affects Bugaboo's ability to accurately detect bugs. There are two reasons for that. First, we do not detect bugs based on *absolute* graph properties, but rather, by detecting graph anomalies — a property *relative* to the emergent communication invariants in a set of graphs. These anomalies manifest themselves even in graphs collected using line addresses, because aberrant communication events (potential bugs) still render themselves as rare edges, just as they do at word gran-

ularity. Second, cache evictions are not likely to have a significant impact on bug detection capability. The reason is that all the operations involved in a schedule-dependent failure typically occur in a short span of dynamic instructions. The communication events not captured due to cache evictions are ones which would have resulted from communication over a long spans of instructions — long enough for data to be evicted from the cache — and are thus unlikely to be of any use in debugging.

4.2.4 Characterization

Our characterization aims to understand the overheads in CACG-HW and assess its performance cost. We did this characterization using the full applications from Table 2.1 as well as a subset of the PARSEC [17] benchmark suite run with their "simlarge" input set. [1]

Benchmark	Misses per 10k Mem Ops			Sources of Overhead in CACG-HW		
	Read	Write	Coherence	M-D Wr / 10k MOp	CT Wr. / 10k MOp	Traps / 10M MOp
blackscholes	212.41	53.93	0.02	266.36	212.55	11.3
canneal	429.33	15.23	0.00	444.57	429.33	6.2
dedup	5.00	0.49	0.05	5.53	5.09	0.1
ferret	33.70	23.96	0.08	57.74	33.78	0.0
fluidanimate	27.15	8.38	0.25	35.78	27.69	1.6
freqmine	137.59	37.90	0.01	175.50	137.68	8.0
swaptions	23.53	98.60	0.51	122.65	25.09	1.5
vips	254.72	67.42	0.05	322.19	254.81	15.5
x264	75.22	28.09	0.00	103.30	75.22	4.4
AGet-MultVar	10.50	0.25	8.39	19.14	18.91	0.0
PBZip2-Order	0.02	0.00	0.00	0.03	0.02	0.0
Apache-LogSz	3.27	3.78	0.03	7.08	3.31	0.0
MySQL-BinLog	129.15	32.85	0.18	162.18	129.41	6.3

Table 4.3: **Characterization of CACG-HW.**

Overheads of CACG-HW

The main sources of overhead in CACG-HW are writes to cache line meta-data, writes to the CT, and traps to software when the CT is full. Column 5 in Table 4.3 shows that the number of meta-data updates is typically less than 200 per 10,000 memory operations, and is as few as 3 in 1 million memory operations. Meta-data updates happen only during cache misses (Columns 2, 3 and 4) and can be fully overlapped with the misses, being completely

[1] We omitted some benchmarks that would not run in our simulator due to memory limitations.

off the critical path. They therefore do not impose any performance cost. The number of CT updates (Column 6) is predominantly less than 35 per 10,000 memory operations. CT updates are done simultaneously with cache coherence transactions and the operation to perform is a simple FIFO buffer insertion, so it imposes negligible runtime overhead.

The most costly overhead is performing a software trap when the CT is full. Column 7 shows that traps happen just a few times per 10 million memory instructions. These events are very infrequent and unlikely to be a major performance problem. Moreover, it is possible to dedicate a spare core to read the communication table, reducing the cost further. During trap handling, writes to the CT are simply discarded. While this may result in a small number of missed graph edges, it enables uninterrupted execution during trap handling, making it effectively a zero-overhead operation. Another way to mitigate the cost of frequent CT traps is to use a larger CT. The downside to this approach is that increasing the size of the CT increases the on-chip buffering requirements, which can be expensive to design and to keep powered.

Several applications stand out with higher costs: `freqmine`, `vips`, `MySQL-BinLog` and `x264`. These applications have 10 to 20 times more edges in their communication graphs and much higher cache miss rate (Columns 2, 3 and 4), indicating that they have more widespread and frequent communication. This ultimately leads to higher frequency traps to read out the communication table, but even for these applications, the frequency of traps is only about 1 per million memory operations, which is unlikely to impede performance.

4.3 Conclusions, Insights, and Opportunities

This chapter described CACG-HW, a hardware architectural mechanism for collecting context-aware communication graphs. CACG-HW collects graphs in a multiprocessor system. CACG-HW leverages existing cache coherence support and per-cache-line meta-data to implement graph collection. Our design eliminates almost all of the software runtime overheads associated with graph collection and our hardware extensions are off the critical path of the execution. CACG-HW's performance is acceptable for production systems.

Insights. The main insight in this chapter is that there is useful information that is *already being tracked* by existing architecture-level mechanisms. This chapter shows how

such information — in this case, inter-processor communication — can be exposed through architectural extensions to an analysis like Bugaboo or Recon.

Opportunities. Many of the opportunities for this work going forward are the same as those in Chapters 2 and 3. One especially interesting direction for future work is studying the role of graph collection in production systems, which is enabled by CACG-HW. In production, graph collection can serve as an underlying mechanism for implementing invariant-guided execution [139], or as support for a failure avoidance mechanism like the one described in Chapter 7.

A shortcoming of this work is the need to approximate communication by monitoring only cache-to-cache transfers. The reason for this limitation is that it is too expensive to keep and update meta-data for all of memory all the time. A profitable direction for research going forward is to look for new ways to keep whole-memory meta-data that are space- and time-efficient.

Finally, another opportunity to go further with the work in this chapter is to take advantage of the information furnished by existing cache coherence mechanisms. Systems already monitor this information. Creating an interface to expose it to analyses more broadly would empower systems. Chapters 5 and 6 begin down this path, by looking at mechanisms that expose coherence to dynamic analyses implemented in architectural extensions.

Chapter 5

ATOM-AID: AVOIDING ATOMICITY VIOLATIONS

Code that fails to enforce necessary atomicity can lead to schedule-dependent failures in production systems. A recent comprehensive study [79] found that atomicity violations accounted for about two thirds of all studied, non-deadlock concurrency bugs. This chapter describes a technique and system support that avoids schedule-dependent failures due to atomicity violations.

The goal of the work in this chapter is to dynamically enforce the atomicity of sequences of operations that were intended to be atomic, but were not made atomic using synchronization. The mechanism we develop works by arbitrarily grouping contiguous sequences of dynamic instructions into *dynamic atomic blocks* ("blocks") that execute atomically with respect to operations in other threads. Like prior work [27, 132, 125], we rely on an execution model in which every instruction executes inside a block. We refer to this style of execution as providing *implicit atomicity* for sequences of operations in the same block.

Implicit atomicity reduces the amount of interleaving of operations executing in different threads, because interleavings can happen only at a block granularity. One of our main findings in this chapter is that implicit atomicity avoids atomicity violations. When a sequence of operations that should be atomic happen to execute together in the same dynamic atomic block, they are guaranteed to execute atomically, whether the programmer correctly implemented the atomicity or not.

We build on this finding and propose Atom-Aid, a new technique for avoiding failures due to atomicity violations. Atom-Aid works by dividing the dynamic instruction stream into blocks, like in an execution model with implicit atomicity. However, rather than establishing block boundaries arbitrarily, as in implicit atomicity, Atom-Aid does so using a heuristic. Atom-Aid's heuristic uses *serializability analysis* to identify regions of code that

should execute atomically. Atom-Aid attempts to establish block boundaries that execute all instructions in each such region in the same block, ensuring their atomicity.

Figure 5.1 shows a simple example: `counter` is a shared variable, and both the read and update of `counter` are inside distinct critical sections under the protection of lock *L*. However, the code is still incorrect as a call to `increment()` from another thread could be interleaved between the read and update of `counter`, leading to incorrect behavior: two concurrent calls to `increment()` could cause `counter` to be incremented only once.

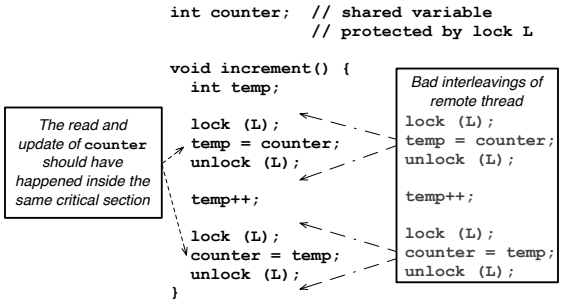


Figure 5.1: **A simple example of an atomicity violation.** The read and update of `counter` from two threads may interleave such that the counter is incremented only once.

The likelihood that an atomicity violation occurs is linked to the number of opportunities for code to interleave in a way that violates the assumptions about atomicity made by the programmer. Figure 5.2(a) shows a sequence of operations that read, increment, and update a shared counter variable called `counter`. The programmer has failed to enforce the atomicity of the read, increment, and update, but for correctness those operations must execute atomically. The figure shows four points in the execution where instructions in the region of code that should be atomic can be interleaved *at a fine grain* by instructions from a different thread, leading to a failure. In contrast, Figure 5.2(b) shows that with the code grouped into atomic blocks, a much smaller set of points in the execution when such interleavings can happen because the operations that were intended to be atomic are executing in the same dynamic atomic block. In these cases, the dynamic atomic block *avoids* the atomicity violation (and hence the failure), despite the fact that the incorrectly written code does not prevent this behavior. The observation that failures can be avoided

by reducing the potential for interleaving of operations in different threads is one of the key contributions of this chapter. In Section 5.2, we explain and analyze this observation in detail.

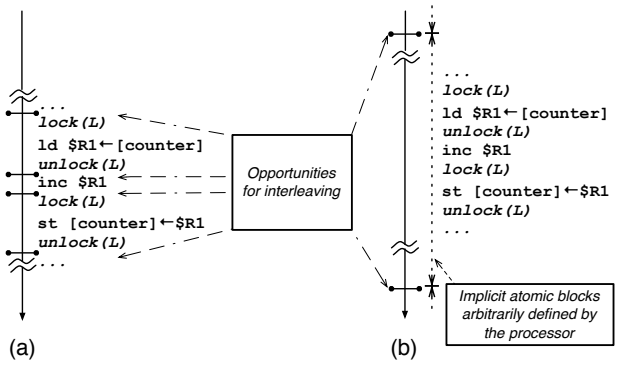


Figure 5.2: **Opportunities for interleaving.** (a) shows where interleaving from other threads can happen in a traditional system. (b) shows where such interleavings can happen in systems that provide implicit atomicity.

This chapter makes several contributions. First, we make the fundamental observation that systems with implicit atomicity naturally avoid some atomicity violations. We analyze and illustrate this observation with a probability study and empirical evidence. Second, we propose Atom-Aid, a novel failure avoidance mechanism using architecture support to detect likely atomicity violations and heuristically determine dynamic atomic block boundaries. Atom-Aid's mechanisms both *detect and avoid* atomicity violations without requiring any human intervention or program annotation. To the best of our knowledge, this chapter describes the first work on dynamically avoiding schedule-dependent failures without requiring global checkpointing and recovery [112, 123, 136]. To aid in debugging and fixing programs that suffer from atomicity violations, Atom-Aid also reports code that is likely to be involved in atomicity violations. Third, we evaluate our technique using buggy code from real applications showing that Atom-Aid is able to reduce the occurrence of a failures due to potential atomicity violations by orders of magnitude.

5.1 Background on Implicit Atomicity

In execution models with implicit atomicity, memory operations in the dynamic instruction stream are arbitrarily grouped into atomic blocks. Recent examples of such systems, like BulkSC [27], Atomic Sequence Ordering (ASO) [132], TCC [56, 131] and Implicit Transactions [125] enforce consistency at the coarse granularity of blocks, rather than individual instructions. These systems provide *implicit* atomicity because blocks do not correspond to anything in the program code – a contrast to the explicit atomic blocks of transactional memory systems. Typically, systems with implicit atomicity take periodic checkpoints (*e.g.*, every 2,000 dynamic instructions) forming blocks of dynamic instructions that appear to execute atomically and in isolation.

Atom-Aid takes advantage of two interesting properties of implicit atomicity. First, implicit atomicity reduces the number of points in a program's execution when instructions in one thread can be interleaved by instructions in another thread. Interleaving can only happen at the granularity of dynamic atomic blocks and the effects of remote threads are visible only at block boundaries. Figure 5.3 contrasts fine-grained interleaving with coarse-grained interleaving. In Figure 5.3(a), interleaving can happen between any instructions (shown on the left side) and there are six possible execution schedules (shown on the right side). In Figure 5.3(b), interleaving opportunities happen only between blocks and there are far fewer possible execution schedules — only two in this example.

The second interesting property is that software is oblivious to the granularity of the atomic blocks in an execution model that provides implicit atomicity. This granularity-independence allows the system to *arbitrarily* define block boundaries and to adjust the size of blocks dynamically without affecting program correctness or the memory semantics observed by the software.

Atom-Aid can be implemented in any architecture that supports implicit atomicity or, more generally, any system that supports forming arbitrary atomic blocks from the dynamic instruction stream. However, for the purpose of illustration, in this work we assume an underlying system similar to BulkSC [27] or TCC [56], in which processors repeatedly execute atomic blocks separated by checkpoints — no dynamic instruction is executed outside an

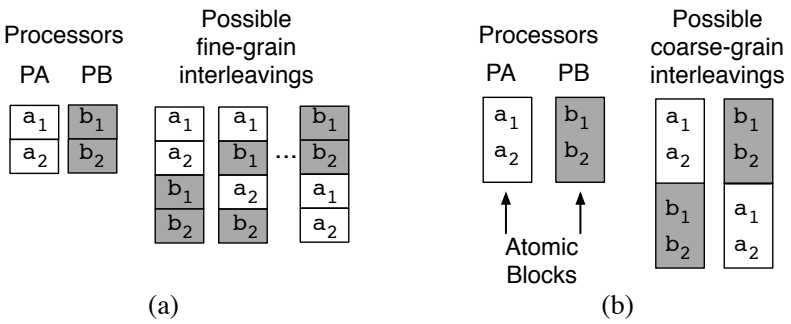


Figure 5.3: **Fine- (a) and coarse-grained (b) access interleaving.** There are six possible interleavings for the fine-grained system and two possible interleavings for the coarse-grained system.

atomic block. To provide a more detailed review of such architectures, we now briefly describe one such system, BulkSC, as its mechanisms naturally provide much of what an implementation of Atom-Aid needs.

5.1.1 BulkSC: A Typical Implicit Atomicity System

Bulk [28] is a set of hardware mechanisms that simplify the support of common operations in environments with multiple speculative threads (or tasks) such as Transactional Memory (TM) and Thread-Level Speculation (TLS). A hardware module called the bulk disambiguation module (BDM) dynamically summarizes the addresses that a task reads and writes into read (R) and write (W) signatures, respectively. A signature is an inexact encoding of addresses following the principles of Bloom filters [19], which are subject to aliasing. Consequently, a signature represents a superset of the original address set. The BDM also includes units that perform signature operations like union and intersection.

BulkSC leverages a cache hierarchy with support for Bulk operations and a processor with efficient checkpointing. The memory subsystem is extended with an arbiter to guarantee a total order of commits. As a processor speculatively executes an atomic block (called a "chunk" in BulkSC), it buffers updates to memory in the cache and generates a Read and a Write signature. When some chunk, i, completes, the processor sends the arbiter a request to commit, together with signatures R_i and W_i. The arbiter intersects R_i and W_i with the W signatures of all the currently-committing chunks. If all intersections are

empty, W_i is saved in the arbiter and also forwarded to all interested caches for commit. Each cache uses W_i to perform bulk disambiguation and potentially abort local chunks in case a conflict exists. Chunks are periodic and boundaries are chosen arbitrarily (*e.g.*, every 2,000 instructions). Forward progress is guaranteed by reducing chunk sizes in the presence of repeated chunk aborts.

5.2 Implicit Atomicity Hides Atomicity Violations

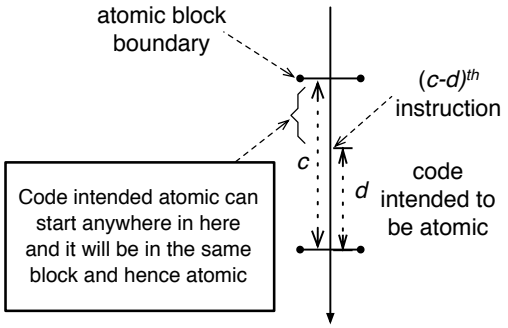
Systems with implicit atomicity reduce the likelihood that an atomicity violation manifests itself by preventing atomicity violations from being *exposed*. An atomicity violation is exposed when instructions that should be atomic execute in *different* atomic blocks and nothing in the code enforces the atomicity of those instructions. When instructions that should be atomic execute in the *same* atomic block – regardless of whether the code enforces the atomicity of those instructions – the potential for an atomicity violation involving those instructions is *hidden*. As a system with implicit atomicity arbitrarily forms dynamic atomic blocks, simply by chance, some atomicity violations are hidden. We call this phenomenon *natural hiding*.

Achieving the same effect of implicit atomicity statically, by having a compiler automatically insert arbitrary transactions in a program, is a challenge. Doing so could reduce performance or prevent forward progress [21]. Neither is a problem in this work because we assume implicit atomicity and systems that support implicit atomicity provide forward progress guarantees [27, 56].

5.2.1 Probability Study

In comparison with conventional fine-grained execution models, implicit atomicity's natural hiding lowers the probability that an atomicity violation is exposed. Our analysis considers a sequence of d instructions that should be executed atomically, constituting a potential atomicity violation. We consider a system with implicit atomicity that uses dynamic atomic blocks constructed of c dynamic instructions. P_{hide} is the probability that all d instructions that should be atomic execute in the same atomic block — *i.e.*, P_{hide} is the probability that the atomicity violation is hidden.

Figure 5.4 illustrates how we derive P_{hide}. If the first instruction in the sequence that should be atomic is within the first $(c - d)$ instructions of a block, then the entire sequence executes atomically within that block, hiding the atomicity violation. With this model, we can express the probability of hiding an atomicity violation as shown in Figure 5.4.



$$P_{hide} = \begin{cases} 0 & \text{if } c < d \\ \frac{c-d+1}{c} & \text{if } c \geq d \end{cases}$$

Figure 5.4: **Naturally hiding an atomicity violation.** The figure shows the boundaries of a sequence of instructions intended to be atomic within dynamic atomic block boundaries. P_{hide} is the probability that the entire sequence executes within the block.

An instruction granularity system effectively executes dynamic atomic blocks including only a single instruction ($c = 1$). In such systems, $P_{hide} = 0$, because, by definition, an atomicity violation involves at least two instructions ($d \geq 2$). This is consistent with the intuition that an instruction granularity system cannot hide atomicity violations. Natural hiding is limited by the length of dynamic atomic blocks. If a sequence of instructions that should be atomic is longer than the number of instructions in an atomic block ($d > c$), then $P_{hide} = 0$ as well. Despite this limitation, implicit atomicity naturally hides some atomicity violations without ever increasing the likelihood that they manifest or changing the program's semantics.

Figure 5.5 shows the probability of hiding atomicity violations for sequences of instructions of varied length as the dynamic atomic blocks size increases, according to the expression of P_{hide} shown in Figure 5.4. As expected, we observe that increasing the block size increases the probability that an atomicity violation is hidden, but that increase is subject to diminishing returns. From our experiments, we observe that typical sequences of instructions that should be atomic tend be around 500 to 750 instructions. Assuming a block size

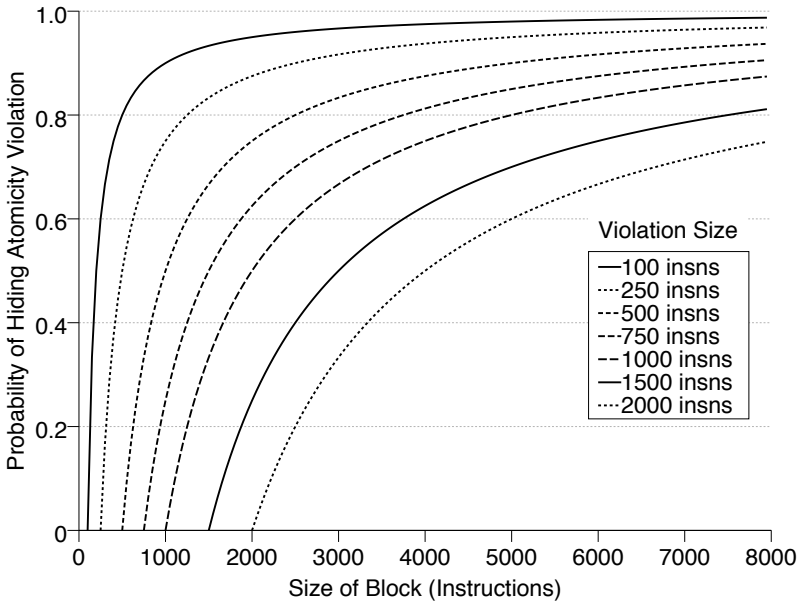


Figure 5.5: **Probability of hiding atomicity violations as a function of dynamic atomic block size.**

of 2,000 instructions, we observe in Figure 5.5 that the expected probability of naturally hiding these typical atomicity violations is 63-75%.

5.3 Actively Hiding Atomicity Violations

As we described in Section 5.1, block boundaries in systems with implicit atomicity are arbitrary and varying their placement cannot affect a program's memory semantics. Atom-Aid takes advantage of the flexibility defining block boundaries by using a heuristic to automatically determine where to place boundaries to further increase the probability of hiding atomicity violations. Atom-Aid's heuristic detects potential atomicity violations. Once a potential atomicity violation is detected, Atom-Aid inserts a block boundary immediately before the first memory access in the sequence of operations involved in the violation. The goal of Atom-Aid's strategy to inserting boundaries is to make all of the instructions in the detected potential atomicity violation execute in the same block. Atom-Aid infers where critical sections should be in the dynamic instruction stream and inserts block boundaries to

make those critical regions atomic. Atom-Aid's boundary insertion strategy is transparent to software and it is oblivious to synchronization constructs that might be present in the code.

5.3.1 Detecting Potential Atomicity Violations Using Serializability Analysis

The key to Atom-Aid is precisely detecting when in an execution a potential atomicity violation may occur. Atom-Aid's heuristic for detecting potential atomicity violations relies on a property of a program's execution called *serializability*. If an execution has no atomicity violations it is *serializable*. A serializable execution is equivalent to some sequential execution of the groups of instructions intended to be atomic by the programmer (a "serialized" execution). In an execution, such regions of instructions may interleave one another and remain serializable if the result of the execution is equivalent to that of some serialized execution.

Conversely, if an execution suffers an atomicity violation, the execution is not serializable — *i.e.*, there is no equivalent serialized execution that produces the same final state. Serializability analysis determines whether interleavings of accesses to a shared variable are serializable. Lu *et al.* [80] used serializability analysis to identify unserializable access interleavings (*i.e.*, potential atomicity violations) detected by their AVIO system. Table 5.1 reproduces the analysis presented in [80]. The column "Interleaving" represents the interleaving in the format $\begin{smallmatrix} A \\ B \end{smallmatrix} \leftarrow C$, where A and B are the pair of local memory accesses interleaved by C, the access from a remote thread. For example, $\begin{smallmatrix} R \\ R \end{smallmatrix} \leftarrow W$ corresponds to two local read accesses interleaved by a remote write access.

Atom-Aid detects potential atomicity violations by monitoring how memory accesses in an atomic block are interleaved by memory accesses from other blocks in the system, as those other blocks complete. When Atom-Aid detects at least two recent accesses (*e.g.*, in the current block) to a variable a by a *local* thread and at least one recent access to a by a *remote* thread, it looks at the types of the accesses involved and applies serializability analysis. If the local accesses were intended to be atomic, the remote accesses interleaved them, and the interleaving was unserializable then an atomicity violation has occurred.

Interleaving	Serializable?	Comment
$\begin{matrix}\mathrm{R}\\ \mathrm{R}\end{matrix}\!\!\leftarrow\!\mathrm{R}$	Yes	—
$\begin{matrix}\mathrm{R}\\ \mathrm{R}\end{matrix}\!\!\leftarrow\!\mathrm{W}$	No	Interleaved write makes two local reads inconsistent.
$\begin{matrix}\mathrm{R}\\ \mathrm{W}\end{matrix}\!\!\leftarrow\!\mathrm{R}$	Yes	—
$\begin{matrix}\mathrm{R}\\ \mathrm{W}\end{matrix}\!\!\leftarrow\!\mathrm{W}$	No	Local write may depend on result of read, which is overwritten by remote write before local write.
$\begin{matrix}\mathrm{W}\\ \mathrm{R}\end{matrix}\!\!\leftarrow\!\mathrm{R}$	Yes	—
$\begin{matrix}\mathrm{W}\\ \mathrm{R}\end{matrix}\!\!\leftarrow\!\mathrm{W}$	No	Local read does not get expected value.
$\begin{matrix}\mathrm{W}\\ \mathrm{W}\end{matrix}\!\!\leftarrow\!\mathrm{R}$	No	Intermediate value written by first write is made visible to other threads.
$\begin{matrix}\mathrm{W}\\ \mathrm{W}\end{matrix}\!\!\leftarrow\!\mathrm{W}$	Yes	—

Table 5.1: **Serializability analysis.** The table shows the analysis and interpretation of each interleaving described in [80].

Even if the interleaving did not occur, if the local accesses should be atomic and some remote accesses executed proximally to them, those remote accesses may interleave the local accesses under some future execution of the same code. Atom-Aid treats interleavings and potential interleavings as potential atomicity violations.

When Atom-Aid finds a potential atomicity violation, it starts monitoring the variable involved in the accesses. When the local thread accesses that variable again, Atom-Aid decides if a block boundary should be inserted. Atom-Aid maintains a history of memory accesses for use in serializability analysis by recording the read and write sets of the most recent local dynamic atomic blocks and recently committed remote blocks.

Figures 5.6 and 5.7 show how Atom-Aid's heuristic is applied to a counter increment example, assuming BulkSC provides implicit atomicity. Atom-Aid maintains the read and write sets of the previously committed blocks: R_P and W_P, respectively. Recall that in BulkSC, processors committing blocks send their write sets to other processors in the system, giving Atom-Aid's analysis the information it needs about what was written recently by

remotely completing blocks (W_{RP}).

In Figure 5.6, processors P1 and P2 are both executing `increment()`. There is a chance that the read and update of `counter` will be atomic due to natural hiding, but in Figure 5.6 that did not happen. The read of the `counter` is inside the previously committed block (P), but the update is part of the currently executing block (C). When `counter` is updated in C, Atom-Aid determines that it was read by the previous local block (`counter` $\in R_P$) and recently updated by a remote processor (`counter` $\in W_{RP}$). This pattern of accesses characterizes a potential atomicity violation.

When Atom-Aid detects this potential atomicity violation, it starts monitoring `counter`. Atom-Aid monitors a variable by making it a member of the *hazardDataSet*, a per-processor set of variables involved in a potential atomicity violation. Later (Figure 5.7), when P1 accesses `counter` again, Atom-Aid detects that `counter` $\in$ *hazardDataSet* and a block boundary should be inserted before the read from `counter` is executed. This increases the chances that both accesses to `counter` will be enclosed in the same block, making them atomic.

While the atomicity violation in Figure 5.6 is exposed, it does not mean it has manifested itself. Also, as will be explained in the next section, even if the atomicity violation is naturally hidden, Atom-Aid is still able to detect it. This shows that Atom-Aid is able to detect atomicity violations before they manifest themselves.

In the following sections, we explain in detail how the detection algorithm works and how Atom-Aid decides where to place block boundaries. We also describe an architecture built around signature operations that implements the mechanisms used by the algorithm.

5.4 Design Overview

There are two main parts to Atom-Aid's design. First, Atom-Aid tracks memory access histories to detect potential atomicity violations. Second, Atom-Aid uses an extension to architecture support for implicit atomicity to determine dynamic atomic block boundaries, aiming to hide atomicity violations and avoid failures.

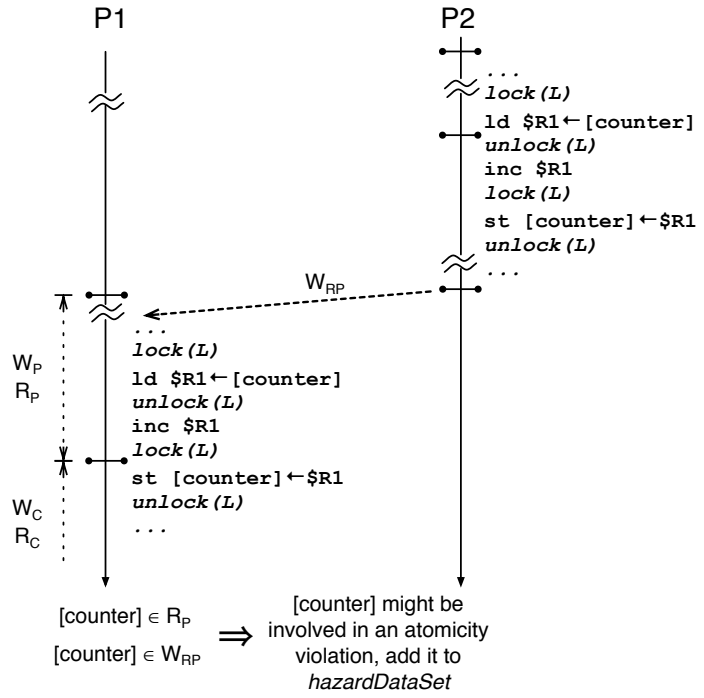


Figure 5.6: **Identifying data involved in a potential atomicity violation.** Atom-Aid discovers that `counter` might be involved in an atomicity violation and adds it to the *hazardDataSet*.

5.4.1 Detecting Potential Atomicity Violations

When Atom-Aid finds two or more accesses by a thread to the same address and one recent access by another thread to that same address, Atom-Aid examines the types of accesses to determine whether they are potentially unserializable. If they are, Atom-Aid treats them as a potential atomicity violation.

Atom-Aid needs to track three pieces of information:

1. The type t (read or write) and address a of the memory operation currently executing

2. The read and write sets of the current and previously committed dynamic atomic block, which we refer to as R_C, W_C, R_P and W_P, respectively

3. The read and write sets of dynamic atomic blocks committed by remote processors while the previously committed local block was executing (referred to as R_{RP} and

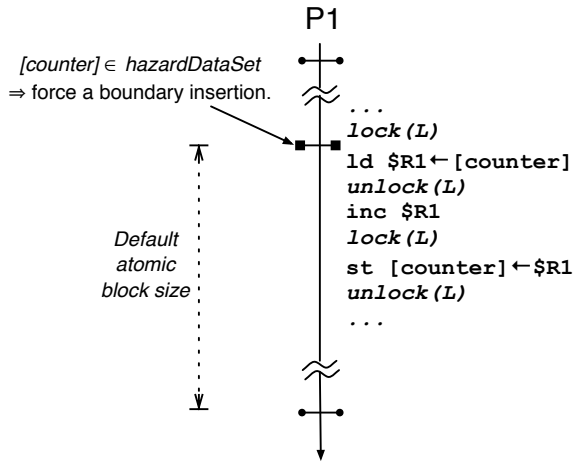


Figure 5.7: **Actively hiding an atomicity violation.** When `counter` is accessed, a block boundary is inserted automatically because `counter` belongs to the *hazardDataSet*.

W_{RP}), together with read and write sets of blocks committed by other processors while the current local block is executing (referred to as R_{RC} and W_{RC}).

Table 5.2 shows how this information is used to determine whether a group of accesses constitute a potential atomicity violation. The first column shows the type of a local memory access, the second column shows which interleavings Atom-Aid tries to identify when it observes this local memory access, and the third column shows a logical expression referring to tracked read and writes sets illustrating how Atom-Aid identifies interleavings of interest.

For example, consider the first two cases: when the local memory access is a read, the two possible non-serializable interleavings are $\overset{R}{R} \leftarrow W$ and $\overset{W}{R} \leftarrow W$. To detect if either of them has happened, Atom-Aid uses the corresponding set expressions in the third column. Specifically, to identify a potential $\overset{R}{R} \leftarrow W$ interleaving, Atom-Aid first checks whether a can be found in any of the local read sets ($a \in R_C \vee a \in R_P$). If it is, Atom-Aid then checks whether a can also be found in any of the remote write sets of a block committed by another processor while either the previous local block was executing ($a \in W_{RP}$) or since the beginning of the current local block ($a \in W_{RC}$). If the condition is satisfied, Atom-Aid identifies address a as potentially involved in an atomicity violation and adds it to

the processor's *hazardDataSet*. Note that this case is not necessarily an atomicity violation because the remote write might not have actually interleaved between two reads. Also, since Atom-Aid keeps only two blocks worth of history, it is only capable of detecting atomicity violations that are shorter than the size of two blocks. This limitation to access history tracking is not problematic, however, because Atom-Aid cannot hide atomicity violations larger than a single dynamic atomic block.

Local Op.	**Interleaving**	**Expression**
Read	R $\leftarrow$W R	$(a \in R_C \vee a \in R_P) \wedge$ $(a \in W_{RC} \vee a \in W_{RP})$
	W $\leftarrow$W R	$(a \in W_C \vee a \in W_P) \wedge$ $(a \in W_{RC} \vee a \in W_{RP})$
Write	R $\leftarrow$W W	$(a \in R_C \vee a \in R_P) \wedge$ $(a \in W_{RC} \vee a \in W_{RP})$
	W $\leftarrow$R W	$(a \in W_C \vee a \in W_P) \wedge$ $(a \in R_{RC} \vee a \in R_{RP})$

Table 5.2: **Cases when an address is added to the *hazardDataSet*.**

5.4.2 Adjusting Dynamic Atomic Block Boundaries

After an address is added to the processor's *hazardDataSet*, every access to this address by the local thread triggers Atom-Aid. If Atom-Aid placed a block boundary right before all accesses that trigger it, Atom-Aid would not actually prevent any atomicity violation from being exposed. To see why, consider Figure 5.7 again. Suppose the address of variable `counter` has been previously inserted in the *hazardDataSet*. When the load from `counter` executes, it triggers Atom-Aid, which can then place a block boundary right before this access. When the store to `counter` executes, it triggers Atom-Aid to again insert a block boundary. If it indeed placed a boundary at that point, Atom-Aid would actually *expose* the atomicity violation, rather than hiding it as intended.

There are other situations in which inserting a block boundary is undesirable. Multiple block boundaries should not be inserted in the case of an atomicity violation involving many accesses. Instead, Atom-Aid should to place just a single block boundary before the

first access. When an address has just been added to the *hazardDataSet* Atom-Aid should not eagerly insert block boundaries. In this case, it is likely that the local thread is still manipulating the corresponding variable, in which case the absence of a block boundary may be beneficial.

To determine whether to place a boundary, Atom-Aid uses a simple policy consisting of two conditions. Figure 5.8 shows this policy in a flowchart. The first condition (1) determines that Atom-Aid never breaks a block into multiple smaller blocks more than once — after a boundary is inserted to avoid a potential violation, the newly created block is always as large as the default block size. The second condition (2) determines that, if Atom-Aid adds any address to the *hazardDataSet* during the execution of a given block, it cannot insert a boundary during that block's execution, breaking it into two blocks.

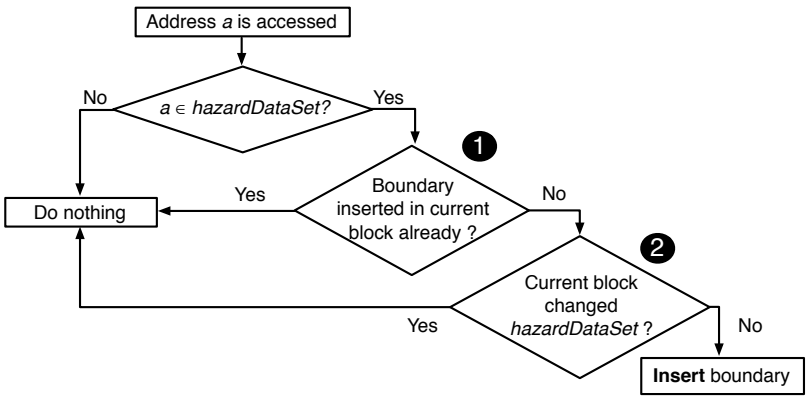


Figure 5.8: **Block boundary insertion logic.** Flowchart showing Atom-Aid's policy for inserting dynamic atomic block boundaries.

5.5 Implementing Atom-Aid with Implicit Atomicity and Hardware Signatures

We based our Atom-Aid implementation on BulkSC because its signatures offer a convenient way of storing blocks' read and write sets. To collect the information required by Atom-Aid's detection heuristic (described in Section 5.4), we add three pairs of signatures to the original BulkSC design. Figure 5.9 shows all the signatures required by Atom-Aid. Signatures R_C and W_C, which hold the read and write sets of the currently executing block,

are used by BulkSC for disambiguation and memory versioning. Signatures R_P and W_P hold the read and write signatures of the previously committed block. When a block commits, R_P and W_P are overwritten with the values in R_C and W_C, respectively. R_{RC} encodes the addresses of all data for which remote read requests were received while the current block is executing. Likewise, W_{RC} encodes the addresses of all data written by remote processors while the current block executes. When a block commits, signatures R_{RC} and W_{RC} are copied into signatures R_{RP} and W_{RP}, respectively, which thus encode the remote operations that happened during the execution of the previous block. If a block is aborted (*i.e.*, due to a memory conflict), only signatures R_C and W_C are discarded, keeping the rest of the memory access history intact.

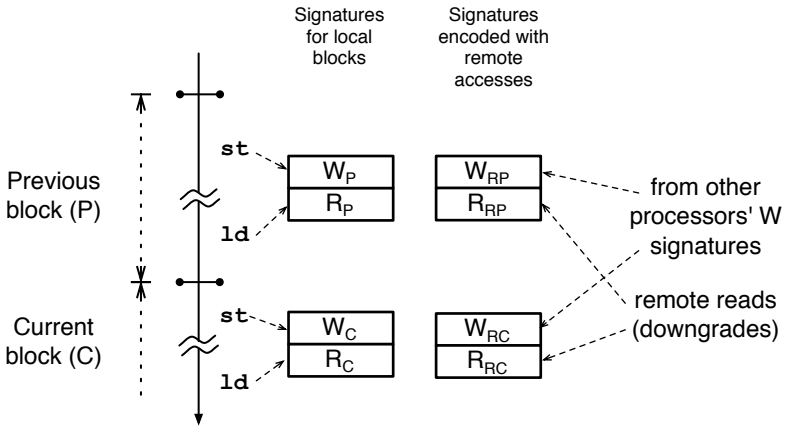


Figure 5.9: **Signatures used by Atom-Aid to detect likely atomicity violations.**

The *hazardDataSet* itself is implemented as a signature as well. When a processor accesses an address, a check to determine if the address is in the *hazardDataSet* is done with a simple membership operation on the signature. The *hazardDataSet* could alternatively be implemented as an extension to the cache tags. An extra bit per cache line tag indicates whether or not the corresponding line address is part of the *hazardDataSet*. When a cache line is accessed, this bit is checked to determine whether the corresponding address belongs to the *hazardDataSet*.

Data Tracking Granularity

Both the signature and cache-based implementations can support word-granularity addresses. With the signature-based implementation, this can be done by simply encoding word addresses as opposed to line addresses. With the cache-based implementation, this can be done by having one bit per word in a cache line to indicate whether the corresponding word is present in the *hazardDataSet*.

The trade-off between these two implementations is one between complexity and effectiveness. While a signature-based implementation is simpler and does not require the address to be present in the cache, it suffers from aliasing (false positives), especially if the *hazardDataSet* contains many data addresses. With the cache-based implementation, there is no aliasing but the implementation of this approach is more complex. In addition, the *hazardDataSet* information of a particular cache line is lost on cache displacements.

Debugging Support

As mentioned earlier, we want Atom-Aid to be useful for debugging tools as well. We envision doing this by making the *hazardDataSet* visible to software and providing a fast user-level trapping mechanism that is triggered at every memory access to addresses in *hazardDataSet*.

Discussion

While we have assumed a BulkSC-like system, other systems that support implicit atomicity can take advantage of the core algorithm in Atom-Aid. For example, in TCC [56] disambiguation is not done with signatures, but write sets are still sent between processors on commit. It is possible to record the information Atom-Aid needs by augmenting TCC with structures to hold the incoming write sets when remote processors commit. It is also possible to use similar structures to hold the read and write sets for previously executed atomic blocks. One key requirement for any implementation of Atom-Aid is that the underlying system providing implicit atomicity guarantees forward progress and the insertion of arbitrary block boundaries does not compromise that guarantee. BulkSC and TCC both

provides this guarantee.

With minor extensions, Atom-Aid can also be implemented in systems that do not support full-fledged implicit atomicity, as long as they provide a mechanism for forming atomic blocks dynamically. Off-the-shelf transactional memory architectures [3] are an interesting platform for studying such an implementation, though we relegate that study to future work.

The run time overhead of the architectural structures required by Atom-Aid is negligible. Membership operations with signatures are very fast because they do not involve any associative search and require only simple logic [28, 117]. As a result, accessing signatures is likely to be much faster than accessing the cache to read or modify data. Also, all accesses to signatures required by both the atomicity violation detection algorithm and the block boundary insertion policy can be done in parallel. In case the *hazardDataSet* is implemented as an extension to the cache tags, it is also unlikely that accessing it will affect performance because both the data and the bit indicating that the corresponding address is part of the *hazardDataSet* can be fetched simultaneously.

Preserving signatures across context switches poses some complexity. The underlying implicit atomicity model is likely to provide a mechanism for preserving read and write sets already — this mechanism can be re-purposed to preserve the additional read and write sets required by Atom-Aid, as well as the *hazardDataSet*. If the cost of preserving these signatures across context switches is excessive in time or space, the signatures can be discarded. While discarding these signatures may lead to some missed detections or boundary insertions, it cannot lead to incorrect behavior or altered program semantics. Furthermore, Atom-Aid is a *best effort* probabilistic failure avoidance mechanism – there are no guarantees that need to be upheld, so trading detection precision for reduced complexity is a reasonable trade-off.

5.6 Evaluation

We have several goals in evaluating Atom-Aid. First, we characterize atomicity violations in a set of bug kernels and full application benchmark programs. Second, we experimentally validate and characterize the ability of implicit atomicity to naturally hide atomicity

violations in our benchmarks. Third, we show that Atom-Aid's block boundary insertion heuristic hides more atomicity violations than natural hiding. Fourth, we characterize Atom-Aid's behavior and evaluate its sensitivity to an implementation that uses signatures, versus a precise implementation. Fifth and finally, we describe the applicability of the information collected by Atom-Aid to the problem of debugging the code that leads to the atomicity violations in our benchmark programs.

5.6.1 Simulation Infrastructure

We model a system similar to BulkSC [27] using the Pin [87] dynamic binary instrumentation infrastructure. Our model includes atomic-block-based execution, using signatures to represent read and write sets, and the mechanisms required by Atom-Aid's algorithm. Unless otherwise noted, the signature configuration used is the same as in [28]. Since the simulator is based on binary instrumentation and runs workloads in a real multiprocessor environment, it is subject to nondeterminism. For this reason, we present all of our results averaged over a large enough number of runs, with error bars showing the 95% confidence interval for the average.

Our simulations determine, for each benchmark, whether a sequence of instructions that should be atomic to avoid a failure is fully enclosed within a dynamic atomic block and the atomicity violation is, therefore, hidden. To determine how often Atom-Aid hides atomicity violations, we explicitly annotated the beginning and end of regions of code that should be atomic. Our simulation model then dynamically checks if these markers fall within the same dynamic atomic block. If so, the corresponding atomicity violation is hidden and it is exposed otherwise. It is important to note that these annotations are not used by Atom-Aid's algorithm in any way — their sole purpose is to evaluate the techniques we propose.

5.6.2 Simulated Workloads

For our experiments, we use two types of workloads: bug kernels and full applications. The purpose of using bug kernels is to generate extreme conditions in which potential atomicity

violations occur more often than in real applications. We also include entire applications (MySQL, Apache, XMMS) to assess how effective Atom-Aid is under realistic execution conditions. For the MySQL runs, we used a multi-threaded variant of the SQL-bench test-insert workload, which performs insert and select queries. For Apache runs, we used the ApacheBench workload. For XMMS, we played a media file with the visualizer on.

We created bug kernels from real applications described in prior work on atomicity violations [49, 80, 136]. We ensured the atomicity violation in the original application remained intact in the kernel version. Wherever possible, we also included program elements that affect timing, such as I/O, to mimic realistic interleaving behavior in the kernel workloads.

Table 5.3 lists the workloads we use in our evaluation. We provide the number of threads each workload uses, the average, minimum, maximum, and standard deviation values of number of instructions in each region that should be atomic. We also include a brief description of each bug.

			Atom. Vio. Size			
Bug Type	**Bug Name**	**Thds**	**Avg.**	**S.D.**	**Range**	**Description**
kernels	Apache-extract	2	973	18.63	909-1014	Kernel version of above Apache log system bug.
	BankAccount	2	85	1.21	81-91	Shared bank account data structure bug.Simultaneous withdrawal and deposit with incorrectly synchronized program may lead to inconsistent final balance.
	BankAccount2	2	2407	1.38	2403-2411	Same as previous, with larger atomicity violation.
	CircularList	2	587	1.88	585-595	Shared work list data ordering bug. Removing, processing, and adding work units to list non-atomically may reorder work units in list.
	CircularList2	2	3593	2.92	3588-3608	Same as previous, with larger atomicity violation.
	LogProc&Sweep	5	278	1.69	272-282	Shared data structure NULL dereference bug. Threads inconsistently manipulate shared log. One thread sets log pointer to NULL, another reads it and crashes.
	LogProc&Sweep2	5	2498	5.55	2489-2514	Same as previous, with larger atomicity violation.
	MySQL-extract	2	239	0.40	239-243	Kernel version of above MySQL log system bug.
	StringBuffer	2	556	0.00	556-556	java.lang.StringBuffer overflow bug [49]. On append, not all required locks are held. Another thread may change buffer during append. State becomes inconsistent.
real	Apache	25	464	0.00	464-464	Logging bug in Apache httpd-2.0.48. Two threads access same log entry without holding locks and change entry length. This leads to missing data or crash.
	MySQL	28	722	8.37	713-736	Security backdoor in MySQL-4.0.12. While one thread closes file and sets log status to closed, other thread accesses log. Logging thread sees closed log, and discards entries.
	XMMS	6	586	6.93	572-595	Visualizer bug in XMMS-1.2.10, a media player. While visualizer is accessing PCM stream data, data in PCM can be changed, or freed, causing corruption or crash.

Table 5.3: **Bug benchmarks used to evaluate Atom-Aid.**

The sequences of instructions intended to be atomic vary in length, by benchmark, from around 80 to around 3,600 dynamic instructions. The violation sizes found in real applications are as large as several hundred instructions, never exceeding one thousand instructions. We believe that the reason these atomicity violations are relatively short is because long atomicity violations are easier to find during testing, since they are bound to manifest more often. The violations found in most bug kernels are similar in size to the real applications. We also used a second, modified version of three kernels in which we added more instructions to the sequence intended to be atomic. Our goal with these modified kernels was to evaluate Atom-Aid with longer atomicity violations (BankAccount2, CircularList2 and LogProc&Sweep2). Note that, for Apache and MySQL, the violation sizes in the full application and the kernel versions are different. This is because in Apache-extract there was additional work in generating random log entries, and MySQL-extract does not use MySQL's custom implementation of `memcpy`.

The real applications we study use a relatively larger number of threads than kernels, ranging from 6 to 28 threads. Most bug kernels use only 2 threads because they are sufficient to manifest the atomicity violation. For the experiments we present in Section 5.6, we simulated each of the bug kernels forty times for each dynamic atomic block size, varying the block size from 750 to 8,000 instructions. We ran each of the real applications five times, with a block size of 4,000 instructions.

5.6.3 Natural Hiding

We validated the probability study in Section 5.2.1 by running the workloads from Table 5.3 in our simulator configured as a system with implicit atomicity but without Atom-Aid. We varied the block size and measured how often sequences of dynamic instructions that should be atomic execute within the same block – *i.e.*, how often the atomicity violation was hidden.

Figure 5.10 shows the percentage of atomicity violations naturally hidden for each of the bug kernels as the block size increases. The lines in the plot correspond to P_{hide} (see Figure 5.4) for the average violation size of each bug kernel shown in Table 5.3. Most experimental data points are very close to the lines derived from the analytical model. This

verifies the accuracy of the model as well as our hypothesis that systems with implicit atomicity naturally hide atomicity violations. While Figure 5.10 does not include data for real applications, the left bar of each cluster in Figure 5.11(b) shows that natural hiding occurs in real applications as well, with a block size of 4,000 instructions. These data also match our analytical model.

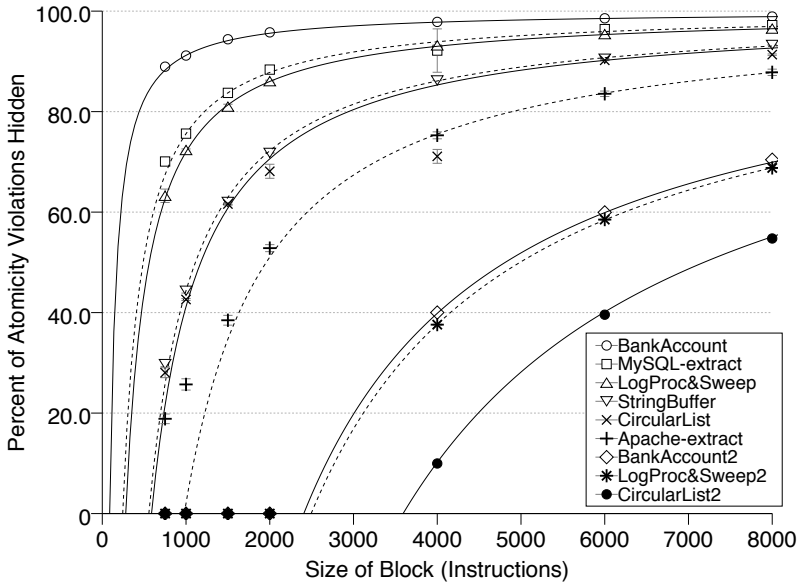


Figure 5.10: **Empirically evaluating natural hiding.** Experimental data on the natural hiding of atomicity violations with implicit atomicity for various block sizes and bug kernels. Points show empirical data, curves show data predicted by our analytical model (P_{hide}).

Implicit atomicity with block sizes as small as 4,000 dynamic instructions naturally hides 70% or more of the atomicity violations for nine of the twelve workloads. The remaining workloads have artificially large atomicity violations that prevent natural hiding at block sizes of 2,000 dynamic instructions or fewer, and keep the probability of natural hiding lower than for other workloads at larger block sizes. As the block size increases, this difference is gradually reduced.

5.6.4 Actively Hiding Atomicity Violations

This section assesses how well Atom-Aid avoids failures by hiding atomicity violations, improving on the already very effective natural hiding of implicit atomicity. Figure 5.11(a) shows the percentage of atomicity violations hidden by Atom-Aid for each bug kernel as the block size increases, whereas Figure 5.11(b) contrasts the hiding effects of Atom-Aid with natural hiding alone, for real applications.

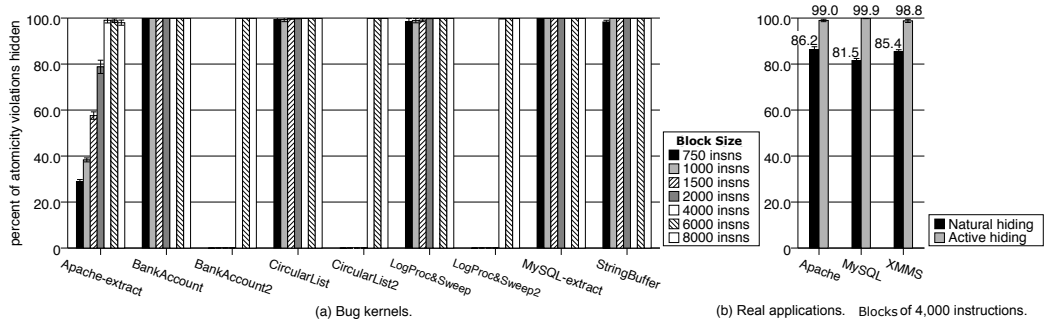


Figure 5.11: **Atomicity violations hidden by Atom-Aid.** Results are averaged over all trials and error bars show the 95% confidence interval.

The data show that Atom-Aid is able to hide very nearly 100% of atomicity violations in our benchmarks, including real applications, with dynamic atomic blocks of only 4,000 dynamic instructions. With even smaller block sizes, Atom-Aid hides the majority of atomicity violation instances. Notable exceptions are Apache-extract and the three bug kernels with artificially longer atomicity violations. As explained in Section 5.6.3, these larger atomicity violations cannot be hidden by blocks shorter than the span of dynamic instructions that should have been atomic. Apache-extract suffers from block boundaries being inserted too early in the execution, which decreases the chance of hiding atomicity violations when smaller block sizes are used. However, the problem disappears when block sizes reach 4,000 dynamic instructions because blocks become large enough to enclose both the access that led to the too-early boundary and the atomicity violation in its entirety.

Overall, Atom-Aid's boundary insertion heuristic is capable of hiding a much larger fraction of atomicity violation instances than natural hiding alone. In comparison with

the off-the-shelf computing systems of today, Atom-Aid reduces the number of exposed atomicity violations by *several orders of magnitude*, hiding more than 99% of atomicity violations in most benchmarks.

5.6.5 Characterization and Sensitivity

Table 5.4 characterizes Atom-Aid's behavior by providing data collected from each of the bug kernels. We only use kernels in this study, as opposed to real applications, because they provide a more controlled environment for our measurements and they run faster in our simulator.

Columns 2 and 3 reproduce data from Figures 5.10 and 5.11, respectively. They show the percentage of hidden atomicity violations with natural hiding and using our boundary insertion algorithm for a block size of 4,000 dynamic instructions. Again, while about 67% of atomicity violations are hidden naturally on average, Atom-Aid's boundary insertion algorithm is able to hide nearly 100% of them.

Column 4 (*% Inserted Boundaries*) shows what fraction of blocks created by the block boundary insertion heuristic as the program executes, while Column 5 (*% Unnecessary Boundaries*) shows what percentage of these additional blocks does not help hide atomicity violations. *% Unnecessary Boundaries* is large for some workloads, showing that Atom-Aid may often insert block boundaries unnecessarily. However, *% Inserted Boundaries* is typically low, so even if it creates many unnecessary blocks, Atom-Aid still adds only a small fraction of all blocks. This implies Atom-Aid is unlikely to have noticeable impact on performance [27].

Columns 6 and 7 illustrate the behavior of Atom-Aid's atomicity violation detection algorithm. Column 6 (*hazardDataSet Size*) shows how many distinct data addresses, at a line granularity, are identified as involved in a potential atomicity violation. Atom-Aid's algorithm selects, on average, only four data items as being potentially involved in an atomicity violation. Column 7 (*# Boundary PCs*) shows how many distinct static memory operations in the code caused Atom-Aid's heuristic to insert a block boundary. On average, it inserts boundaries at only three code points. These results show that Atom-Aid selectively

identifies data addresses and points in the program that are potentially involved in atomicity violations. These can be reported to a programmer who in turn has reasonably precise information about the potential atomicity violation and can use it to debug the application. We further explore this aspect of Atom-Aid in Section 5.6.6.

	Natural	Signature-Based Atom-Aid					Exact Atom-Aid		
	Hiding		% Insert	% Unnec.	*hazardDataSet*	# Bound.		% Insert	% Unnec.
Benchmark	**% Hide**	**% Hide**	**Bound.**	**Bound.**	**Size**	**PCs**	**% Hide**	**Bound.**	**Bound.**
Apache-extract	75.77	99.03	4.4	79.4	5	3	99.94	1.1	16.7
BankAccount	97.84	99.99	12.5	75.2	4	3	99.99	6.4	50.4
BankAccount2	39.6	100.00	11.7	74.8	4	3	100.00	6.1	49.6
CircularList	71.14	99.95	12.5	0.0	2	2	99.95	12.5	0.0
CircularList2	9.92	99.90	11.1	0.1	2	2	99.90	11.1	0.1
LogProc&Sweep	93.14	99.88	12.4	0.2	11	4	99.89	12.4	0.4
LogProc&Sweep2	37.64	99.73	11.0	0.1	2	2	99.78	11.0	0.1
MySQL-extract	91.89	100.00	18.8	46.1	3	6	100.00	18.7	45.6
StringBuffer	86.21	100.00	6.2	0.0	3	2	100.00	6.2	0.0
Average	67.02	99.83	11.2	30.6	4	3	99.94	9.5	18.1

Table 5.4: **Characterizing Atom-Aid.** The table shows data for both the signature and non-signature implementations.

So far, we have discussed data on an implementation of Atom-Aid that exclusively uses hardware signatures for disambiguating accesses performed during block execution, detecting block interleaving, and maintaining the *hazardDataSet*. *Exact Atom-Aid* corresponds to the behavior of a non-signature based implementation of Atom-Aid. For that, all signatures in the design presented in Section 5.5 are simulated ideally as unlimited size exact sets — there is no aliasing when detecting potential violations or when determining if a memory address is in the *hazardDataSet*. We present these results in the group of columns entitled *Exact Atom-Aid* in Table 5.4. The behavior of the exact implementation of Atom-Aid would be similar to the behavior of an implementation that uses cache tag extensions as a way of keeping the sets of addresses.

First and most important, *% Hide* for the exact implementation (Column 8) is nearly the same as *% Hide* for the signature-based implementation (Column 3), showing that the impact of signature imprecision on the effectiveness of Atom-Aid is negligible. *% Inserted Boundaries* (Columns 4 and 9) is, on average, higher for the signature-based Atom-Aid, because aliasing in signatures causes boundaries to be inserted more frequently. However,

the difference is small. The difference in the frequency of boundary insertions is also reflected in the percentage of unnecessary boundary insertions (Columns 5 and 10), which is significantly lower for *Exact Atom-Aid*.

5.6.6 Debugging Discussion

Showing that Atom-Aid is able to hide almost all atomicity violations demonstrates that the algorithm inserts block boundaries in appropriate places. Atom-Aid is also able to report the program counter (PC) of the memory instruction where boundaries were inserted. These places in the program are the boundaries of potentially incorrectly specified critical sections so they can be used to aid the process of locating bugs in the code. While a detailed analysis of a complete debugging tool is outside of the scope of this work, we were able to use the feedback from Atom-Aid to locate the code for the bugs in MySQL and Apache used in past work on bug detection [80, 136], and to detect a bug in XMMS not studied in the prior literature.

We used the following process to locate bugs: (i) collect the set of PCs where block boundaries were inserted; (ii) group PCs into the *line of code* and *function* in which they appear; and, finally, (iii) traverse the resulting list of functions, from most frequently appearing to least, examining the lines of each function, from the most frequently appearing line to least. Using this process, we were able to locate a code point related to each of the bugs in our evaluation by inspecting a relatively small number of points in the code.

Table 5.5 shows some data on our experience finding atomicity violations in real applications. The first group of columns (*Program Totals*) shows the total number of files, functions, and lines of code for the entire application. The second group (*Boundary Insertion Points*) shows the number of files, functions, and lines of code for which Atom-Aid inserted block boundaries while the application executed. The third group (*# of Inspections*) shows the number of files, functions, and lines of code we had to inspect before we located a bug.

For Apache, only 85 lines of code in 6 files needed to be inspected to locate the bug. For MySQL, the number is larger (more than 300), but MySQL has a larger code base, with

almost 400,000 lines of code. We identified a bug in XMMS that was not previously known[1] after inspecting only 9 lines of code. Overall, the information provided by Atom-Aid is useful in directing the programmer's attention to the right region of code, even if using a simple heuristic like the one we present here. However, more sophisticated techniques, like those in Chapter 3, result in even more effective debugging.

	Program Totals			Boundary Insertion Points			# of Inspections		
Program	**Files**	**Func.**	**Lines**	**Files**	**Func.**	**Lines**	**Files**	**Func.**	**Lines**
Apache	729	3361	290k	52	206	956	6	8	85
MySQL	871	15231	394k	44	228	681	27	84	353
XMMS	268	1368	81k	7	23	42	2	4	9

Table 5.5: **Characterization of the bug detection process for real applications using Atom-Aid.**

5.7 Conclusions, Insights, and Opportunities

This chapter described a mechanism for avoiding atomicity violations. We showed how existing coarse-grained execution models that provide implicit atomicity avoid some atomicity violations by default. We showed that serializability analysis applied to parts of a program's execution history can identify likely atomicity violations. Combining implicit atomicity with serializability analysis, we developed Atom-Aid, a mechanism that more precisely avoids atomicity violations, than implicit atomicity alone. We described how Atom-Aid can be implemented in a straightforward way in an architecture that provides implicit atomicity. We then evaluated our system and showed it is effective at avoiding failures and can provide useful debugging information.

Insights. There are several major insights that we gained in developing Atom-Aid. First, in this work, we discovered that it is possible to perturb a shared-memory multi-threaded execution without breaking the semantics of the program executing. Atom-Aid uses this technique to avoid failures and lays the groundwork for other techniques in Chapters 6 and 7. Second, we discovered the phenomenon of "natural hiding" of atomicity violations by systems with implicit atomicity. Third, we found it possible to build a system

[1]The XMMS project leads were contacted regarding the bug. However, no feedback was ever received.

that is effective at avoiding failures, but also provides debugging benefit, making it useful during both development and in production systems.

Opportunities. Atom-Aid is largely a starting point. We showed that avoiding failures is possible by perturbing execution schedules. One major limitation of our approach is the reliance on implicit atomicity. There are no commercial systems with implicit atomicity. Chapter 6 describes a different architecture that is similar to Atom-Aid that obviates the need for implicit atomicity, instead relying on transactional memory support alone.

A second limitation of this work is the inability of this technique to handle atomicity violations involving multiple different variables. The serializability analysis we use is limited to looking at a single variable, but many violations involve groups of data. Chapter 6 generalizes the analysis used in this chapter to overcome this limitation.

Finally, the mechanism for avoiding failures used in this chapter is limited in that it cannot deal with arbitrary schedule-dependent failures (*e.g.*, ordering violations). The work in Chapter 7 addresses this shortcoming by using a different theoretical framework that covers schedule-dependent failures in general.

Chapter 6

COLORSAFE: AVOIDING MULTI-VARIABLE ATOMICITY VIOLATIONS

The previous chapter described Atom-Aid, a system for dynamically detecting and avoiding failures in concurrent programs that result from *atomicity violations*. Atom-Aid, like most prior work on detecting and debugging atomicity violations [136, 80, 44], focused on situations involving a series of accesses to a single variable that were intended to be atomic. Focusing on only single-variable violations leaves techniques fundamentally limited in helping with a wide variety of bugs.

One reason that prior work has focused on single variable situations is that the complexity of the state required and number of cases to be considered when identifying multi-variable atomicity violations grows quickly with the number of variables involved. Also, until recently it was unclear that multi-variable bugs are common. However, recent prior work [78, 79] finds that multi-variable concurrency bugs show up with troubling frequency: roughly one third of atomicity violations involve multiple variables. Errors involving multiple variables are likely to be more difficult for programmers to understand and fix than those involving a single variable. Their frequency and difficulty make multi-variable atomicity violations an important problem to solve.

In this chapter, we develop a general solution that attacks both single- and multi-variable atomicity violations using a single set of simple architectural mechanisms. The mechanisms we develop are useful useful during development for detecting and debugging programming errors that lead to atomicity violations and also useful in production for avoiding failures due to atomicity violations. Like Atom-Aid, our technique works by identifying potential atomicity violations using serializability analysis. The key observation that differentiates this work is that we can create sets of variables that we call colors and apply a specialized serializability analysis that considers colors, rather than individual variables. Using a color-

based serializability analysis, our technique is able to detect and avoid single- and multi-variable atomicity violations.

Our approach builds on the insights of prior work: we use the concept of assigning colors to data from data-centric synchronization [26, 126], we use a variant of the serializability analysis developed for bug detection [80, 136], and we use a variant of the failure avoidance mechanism that we developed in Atom-Aid [85]. We combine these techniques in a novel way in a system that we call ColorSafe. By grouping variables into colors, our detection mechanism considers sets of variables, rather than individual variables. Detecting interleavings of accesses to same-colored data that should have been atomic enables our mechanism to detect both single- and multi-variable atomicity violations. By targeting both classes of atomicity violations, ColorSafe is more general than prior work that focused on single-variable violations only. ColorSafe avoids failures due to these atomicity violations by preventing unintended interleavings using dynamic atomic blocks. In this chapter, we call these blocks *ephemeral transactions* because, unlike Atom-Aid, which required complex architectural support for implicit atomicity [56, 27, 32], ColorSafe requires only a mechanism to dynamically start and end atomic blocks, like standard transactional memory support.

ColorSafe has two modes of operation: *debugging mode* and *deployment mode*. Debugging mode collects more information than deployment mode and uses a stricter criterion for detecting atomicity violations with fewer false positives. Deployment mode provides dynamic bug avoidance with higher recall and lower precision than debugging mode. The key to deployment mode is its use of a less strict criterion for detecting potential bugs that has fewer false negatives at the cost of more false positives. By having these two complementary modes, ColorSafe is useful throughout the entire software lifecycle: during development and after deployment. Providing utility for the lifetime of a system makes adding these mechanisms to a processor more compelling to processor manufacturers. Moreover, the same mechanisms are used to support both modes.

6.1 Multi-variable Atomicity Violations and Serializability

This section reviews atomicity and serializability, which were first introduced in Chapters 1 and 5 and connects these concepts to multi-variable atomicity violations.

Consider the example shown in Figure 6.1(a). The shared variable `ctr` is being incremented by two threads simultaneously, which may lead to an atomicity violation: the write from Thread 2 can interleave with the read and write from Thread 1 and cause a counter increment to be lost. The read and write of `ctr` should have been atomic. This example shows a a single-variable *atomicity violation*, because it involves only accesses to `ctr`. Additionally, this interleaving is *unserializable*: assuming each thread's two accesses to `ctr` should execute as an atomic block, there is no sequential execution of those blocks that produces the same final state as the interleaved execution.

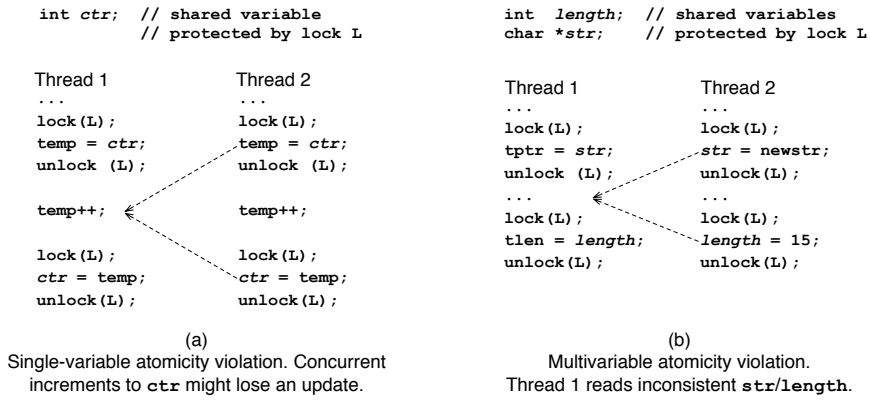


Figure 6.1: **Example atomicity violations.** (a) shows a single-variable violation and (b) shows a multi-variable violation. The example in (b) was distilled from https://bugzilla.mozilla.org/show_bug.cgi?id=73291.

Figure 6.1(b), shows a *multi-variable* atomicity violation. Two shared variables, `str` and `length`, are used to express related properties of a string. After Thread 1 reads the `str` pointer, Thread 2 updates both variables. Thread 1 subsequently reads a value of `length` inconsistent with the value of `str` that it read. The inconsistency could later lead to a crash or silent data corruption failure. Note that, if the regions of code accessing each individual variable are considered separately (*e.g.*, read and write of `str`), the example does *not* show an atomicity violation. Conversely, if the updates to `str` and `length` are considered together as a unit, the atomicity violation is clear: two reads by Thread 1 interleaved by writes in Thread 2. Importantly, these accesses are unserializable with respect to the group containing `str` and `length`, because no serial execution of the threads' code regions would produce the same result as the interleaved one.

To ensure that the examples in Figure 6.1 do not experience failures, the programmer should have enclosed the groups of memory operations intended to be atomic in a single critical section. In prior work, Vaziri *et al.* [126] provide a serializability analysis of code regions including accesses involving two different variables. In ColorSafe, we develop a generalization of serializability analysis that detects single- and multi-variable atomicity violations.

6.2 ColorSafe: Detecting and Avoiding Multi-Variable Atomicity Violations

Grouping variables into colors enables detection of multi-variable atomicity violations. The algorithm ColorSafe takes advantage of data coloring to detect and dynamically avoid single- and multi-variable atomicity violations.

6.2.1 Leveraging Data Coloring

Detecting multi-variable atomicity violations is challenging because conventional serializability analysis consider accesses to a single memory location only. Instead, a multi-variable analysis must consider interleavings of accesses to different data. A multi-variable atomicity violation is depicted in Figure 6.1(b). Thread 1 is performing separate updates of the `str` and `length` variables that should occur atomically together. When Thread 2 interleaves its updates between Thread 1's operations – an atomicity violation – the final values of `str` and `length` are inconsistent. To apply the serializability-based reasoning from prior work [85, 80] to this example, we must consider `str` and `length` as a single unit of data to detect the atomicity violation. This concept of considering groups of variables together, instead of single variables alone, is the cornerstone of ColorSafe.

We propose to associate *colors* with shared variables, giving related variables the same color. ColorSafe monitors interleavings of accesses to colors to determine whether they are serializable the same way AVIO [80] and Atom-Aid [85] did for interleavings of accesses to individual memory locations. In Figure 6.1(b), `str` and `length` would be given the same color since they are semantically related. A serializability analysis that considers colors shows that this example consists of two reads interleaved by at least one remote write to the same color, which is unserializable.

In Table 6.2, we enumerate possible multi-variable unserializable interleavings. Note that Cases 1-4 are unserializable from a single-variable point of view as well. Case 5 extends the serializability analysis from prior work because it is unserializable only if multiple variables are involved. To understand Case 5, consider the example in Figure 6.3. The writes from Thread 2 interleaved with the writes from Thread 1, leaving the consistency between `length` and `str` compromised: `str` may point to `ptr2` but `length` will have value 10. By assigning `str` and `length` the same color, ColorSafe's extended serializability analysis deems the pictured interleaving serializable.

Case	Interleaving	Description
1	R ←W R	The interleaving write might make the second read read inconsistent data.
2	R ←W W	The second write might write data based on stale or inconsistent data.
3	W ←W R	The read might get data from the interleaving write and therefore get inconsistent data.
4	W ←R W	The interleaving read might get inconsistent data.
5	W ←W W	The interleaving write may leave the color inconsistent (Figure 6.3).

Figure 6.2: **Unserializable color access interleavings.**

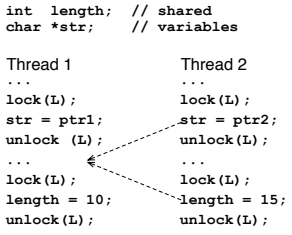


Figure 6.3: **Example of a unserializable color interleaving.** The example corresponds to case 5 in Table 6.2. `str` and `length` are left mutually inconsistent.

Grouping semantically related variables into colors and performing serializability analysis detection based on accesses to colors, simplifies the detection of multi-variable atomicity violations. Coloring data effectively reduces the high complexity of detecting multi-variable atomicity violations [78, 126] to the lower complexity of detecting single-color (or single-variable) atomicity violations.

Coloring Data. Data can be colored manually or automatically. Manual coloring requires the programmer to annotate which data are semantically related to one another. Manual coloring therefore is likely to yield more precise information about relationships between data. Manual coloring requires source code annotations, by which the programmer conveys the semantic relationship between variables. Grouping related data manually has been used in past work to associate synchronization constraints with data (*e.g.*, atomic sets [126],

coloring [26], and locking discipline [92]) but, to our knowledge, never to detect and avoid bugs.

Automatic coloring does not require programmer effort, but is likely to less accurately capture relationships among data. There are many ways of automatically coloring data. Simple strategies include assigning the same color to all fields of a `struct`, to all blocks of memory allocated by the same `malloc()` call, or even entire object instances if using object-oriented languages. Exhaustively exploring different coloring techniques is outside the scope of this work. We do, however, explore both manual coloring and automatic coloring to give the reader a flavor of the differences. We evaluate one automatic coloring technique by giving the same color to data allocated together ("malloc coloring"). Past work has addressed ways of finding correlations between variables [78], but it did not address atomicity violations. Such work is complementary and ColorSafe can use its correlations to color data.

6.2.2 Detecting Unserializable Color-Access Interleavings

ColorSafe detects atomicity violations by detecting unserializable interleavings of regions of code that are likely to have been intended to be atomic. If two regions of code were intended to be atomic, they interleave their operations in an execution, and that interleaving is unserializable, then an atomicity violation is occurring and may lead to a failure. It is important to note that unserializable interleavings are not necessarily manifestations of atomicity violations. However, for an atomicity violation to manifest itself, it is necessary that an unserializable interleaving happens. Moreover, unserializable interleavings, especially in a short window, are strong indicators of atomicity violation bugs [80, 85]. Hence, we detect likely bugs by detecting unserializable interleavings.

In this section, we explain how ColorSafe detects actual unserializable interleavings in debugging mode. The next section explains its differences from deployment mode. The atomicity violation detection mechanism used by ColorSafe has three components: (1) a history of accesses performed by the local processor (*local history*); (2) a history of accesses performed by remote processors (*remote history*); and, (3) a set of rules that determine whether the interleaving of accesses in the history is serializable.

Since our goal is to detect whether accesses to colors are serializable, ColorSafe's access histories record an ordered history of accesses including the color of the accessed data. This means the data addresses of memory accesses need to be translated to colors before accesses can be inserted into the history. Local accesses are inserted into the local history as the local processor performs them. Remote access histories are built by monitoring remote memory accesses to data recently shared between threads. Section 6.3.2 describes how remote access monitoring can be performed efficiently in an architecture implementation by monitoring cache coherence messages. Local and remote histories are kept separately and ColorSafe retains some information about the relative order of groups of local and remote accesses.

To detect atomicity violations, ColorSafe assumes accesses in its histories constitute regions intended to be atomic. Under this assumption, ColorSafe then determines whether any unserializable interleaving of those regions in the local and remote histories exists according to Table 6.2. Figure 6.4 shows an example of this process using the same code from Figure 6.1(b), which is reproduced in Figure 6.4(a) and contains a multi-variable atomicity violation involving two variables. Variables `str` and `length` are both colored *RED*. The numbers in the dark circles denote the order of accesses. The accesses are inserted into their corresponding history as they happen (Figure 6.4(b)). Accesses 2 and 3 performed by Thread 2 are inserted as writes into Thread 1's remote history (2' and 3'). As soon as access 4 is performed, ColorSafe detects that the accesses in the history are unserializable, matching Case 1 in Table 6.2 (two reads interleaved by a write).

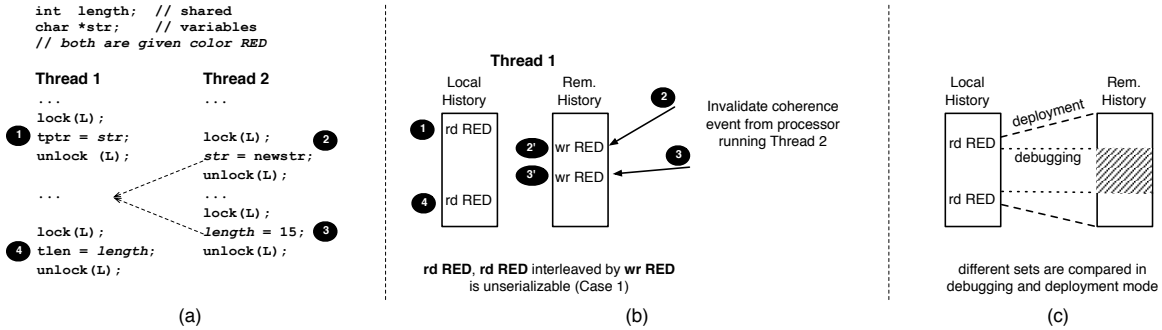


Figure 6.4: **Overview of how ColorSafe detects multi-variable atomicity violations.** The numbers in the dark circles denote the order of events happening simultaneously in (a) and (b).

Recent work [79, 85] shows that to detect atomicity violations histories can be fairly small, on the order of tens of thousands of instructions at most. This is intuitive because the longer the distance between operations that should have been atomic, the greater the chance that the bug manifests itself during testing. Therefore, the hard bugs that escape testing tend to be the ones that occur in a short window.

6.2.3 Debugging vs. Deployment Mode

The previous section described what we call *debugging mode*, as it soundly detects interleavings that are definitely unserializable. In *deployment mode*, ColorSafe attempts to dynamically avoid atomicity violations by (1) detecting when an atomicity violation *may* happen, and (2) dynamically starting an ephemeral transaction to prevent an unserializable interleaving from happening.

In deployment mode, we aim to detect *potentially* unserializable interleavings that are related to *potential* atomicity violations. The avoidance mechanism can then be triggered before an unserializable interleaving actually happens, to prevent it from happening. We relax the criterion used to detect unserializable interleavings in debugging mode such that it covers interleavings that could potentially happen in a future execution of the same code. A *potentially unserializable interleaving* is one involving a pair of operations in the local history that, if interleaved by some access in the remote history, would have been unserializable according to Table 6.2. For example, consider the scenario in Figure 6.4: if the remote writes to *RED* had happened anywhere in the remote history window, even if they did not actually interleave with the reads from *RED* in the local history, this would be detected as a potentially unserializable interleaving. Figure 6.4(c) illustrates the difference between debugging and deployment mode.

The intuition behind this definition of a potentially unserializable interleaving is that, because ColorSafe observed an unserializable interleaving that almost happened, it could observe the actual interleaving at a future point in the execution. This could result in an atomicity violation manifesting itself, which is what ColorSafe is trying to avoid. It is possible that potentially unserializable interleavings are, in fact, just benign accesses.

In this case, the only effect is that ColorSafe initiates unnecessary bug avoidance actions. While this is not a correctness problem and typically not a performance problem either, the system provides hooks to the programmer to disable avoidance actions in performance sensitive parts of the code.

Once a potentially unserializable interleaving is detected, the color of the data accessed is inserted into a set called the *HazardColorSet*. From then on, all accesses to data whose color is in the HazardColorSet trigger an ephemeral transaction of a finite size. The ephemeral transaction will make the short period of the execution beginning with these accesses appear to execute atomically and in isolation, effectively preventing any unwanted interleaving with remote accesses from happening in the meantime. The goal is that this ephemeral transaction will begin with the first instruction of an atomicity violation and be long enough to cover all local memory accesses involved in the violation, consequently preventing its manifestation.

Ephemeral transactions are transactions dynamically inferred by ColorSafe and do not correspond to any program annotation. It is important to point out that the ephemeral transactions inserted by ColorSafe cannot, in any way, break the semantics of the program, since the resulting interleaving of accesses with ephemeral transactions is still a valid interleaving with respect to the program semantics. Section 6.3.4 provides details.

6.3 Architectural Support

ColorSafe needs four basic architectural mechanisms: support for data coloring (Section 6.3.1); histories of recent memory accesses, in terms of colors (Section 6.3.2); a means of detecting unserializable interleavings based on the access histories (Section 6.3.3); and, for bug avoidance, a way of maintaining the set of colors involved in unserializable interleavings together with support for ephemeral transactions (Section 6.3.4).

6.3.1 Support for Data Coloring

ColorSafe represents the color of data items as meta-data (memory tags). There have been several proposals to support memory tagging for various purposes, such as security and information flow tracking [34, 142] and as support for new programming models [26]. To

support ColorSafe, we chose a design similar to Colorama [26], which is based on the Mondrian Memory Protection scheme [133]. Mondrian provides an efficient way to associate protection information with arbitrary regions of memory by using a hierarchical multilevel permissions table. ColorSafe uses the same structure but stores ColorIDs instead of permission information. We call this table the Multilevel Color Table. Based on the number of colors required in the applications used in our experiments, we used a 12-bit ColorID field.

The Multilevel Color Table resides in memory and is accessible by all processors. Its ranges of addresses are expanded to keep the ColorID information at the desired granularity (word, line, page, etc.). The Color Lookaside Buffer (CLB) directly caches coloring information from the Color Table to provide fast lookup. To look up an address, the processor checks the CLB. In case of a miss, the processor fetches the entry from the Multilevel Color Table in memory. Software can update color information in user-mode by writing to the Multilevel Color Table. When the color table is written, the CLB needs to eventually be updated, but not immediately. ColorSafe can tolerate this transient color information incoherence, since this will not affect program semantics in any way.

Note that there are alternatives to providing support for data coloring. For example, Loki [142] proposes a multi-granular tagging mechanism, in which tags can be associated with whole pages and, only when necessary, expanded to individual words to provide fine-grain tagging. Such a scheme would also be adequate for our purposes. Yet another alternative would be to add a ColorID field on a per cache line basis. We opted not to do this for three reasons: (1) we want to allow arbitrary coloring without forcing the user to adjust data layout; (2) the Multilevel Color Table is more space efficient; and, (3) we did not want to alter sensitive structures in the memory hierarchy.

6.3.2 Color Access Histories

In ColorSafe, each processor stores information about the recent history of color accesses in a *history buffer*. A history buffer holds four types of histories: (1) local read, (2) local write, (3) remote read, and (4) remote write.

ColorSafe keeps tens of thousands of instructions worth of history. Therefore it needs

a resource-efficient way of keeping them, as we cannot use a searchable FIFO with tens of thousands of items. We chose to encode the color of accesses in bloom-filter-based signatures [28]. This way, each of the four history types is a *signature file* organized as a FIFO queue, in which each signature is a superset hash-encoding of ColorIDs of memory accesses for an arbitrary number of dynamic instructions. Since only colors, instead of individual addresses, are recorded into signatures, the amount of imprecision (aliasing) in the signatures is low. Figure 6.5(a) shows a color access signature file.

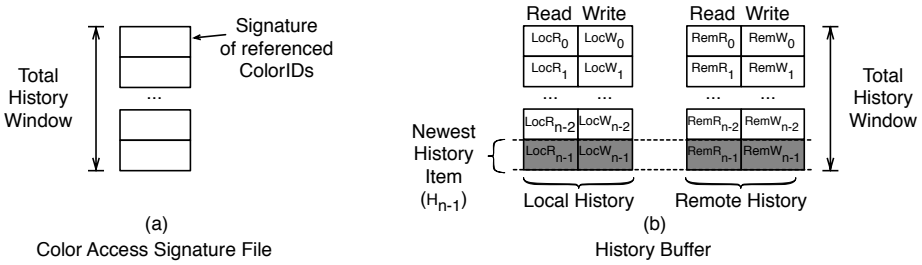


Figure 6.5: **Keeping color access history.**

ColorSafe divides the execution of a program into *epochs*, arbitrary length sequences of consecutive dynamic instructions (*e.g.*, 400). A *history item* is a set of four signatures (one of each history type) that contains color accesses collected during an epoch. Figure 6.5(b) shows a complete history buffer, which is a set of history items that covers the last n epochs of execution. When an epoch ends, both local and remote accesses start being encoded in the next history item. In summary, a history item H_i consists of a Local Read signature $LocR_i$, a Local Write signature $LocW_i$, a Remote Read signature $RemR_i$, and a Remote Write signature $RemW_i$.

Note that ColorSafe sacrifices information about the relative order of operations within a history item. Information about the relative order across history items is preserved, though. The trade-off in determining the history item granularity (assuming fixed total history window) is one of precision versus cost. Smaller history items (finer granularity) improve precision by preserving more relative order information and suffer from less signature aliasing. Larger (coarser-grain) history items use less storage and comparison logic, since fewer history items are necessary and consequently fewer intersection operations need to be performed.

Collecting Local Access Information. Local access information can be easily obtained. ColorSafe uses the mechanism described in Section 6.3.1 to look up the ColorID for each load and store issued locally. The resulting ColorID is then encoded in the appropriate local read and write signatures for the current history item. When an epoch completes, accesses start being inserted in the next history item.

Collecting Remote Access Information. The ColorSafe atomicity violation detection algorithm requires processors to monitor remote memory accesses that access data that was recently shared. ColorSafe monitors such remote memory accesses by monitoring cache coherence messages. Read requests and invalidations signify that a remote read or write has occurred. When such a coherence message is received, a corresponding access is inserted into the remote history.

Recording remote color accesses requires minimal additional cache coherence protocol support. To collect color information for remote accesses, we augment coherence requests without affecting coherence protocol functionality in any way. On a read miss, the processors retrieve the color information of the data being accessed (actual referenced address, as opposed to block address) and appends it to the coherence request sent to potential sharers. When a processor receives a coherence request from a remote processor, it adds the ColorID in the request to its current remote read signature. Likewise, an invalidate request generated by a write miss or a write on shared miss is augmented with a ColorID. Receiving processors add the ColorID to their current remote write signature. ColorSafe needs color information only for accesses that cause inter-processor communication, and so it is sufficient to piggyback on coherence protocol messages.

Relying on coherence messages means that ColorSafe detects atomicity violations involving only data that has been shared and cached by different processors shortly before or during violation. Atomicity violations essentially involve data that is shared by different processors, so there is nothing lost by not monitoring accesses to unshared data. The restriction that sharing must have occurred shortly before or during the atomicity violation is also not problematic. An eviction from an involved processor's cache may result in a coherence message not being sent for some accesses. However, evictions are not likely to be a problem because atomicity violations tend to involve regions of code that are relatively

short [85, 80]. Data are likely to remain in the cache for the entire violation.

6.3.3 Detecting Unserializable Interleavings

ColorSafe detects unserializable interleavings by intersecting signatures in the history buffer. A signature intersection is a simple bit-wise *AND*. For example, suppose we want to detect whether Case 1 in Table 6.2 happened in the history buffer. Using the symbols in Figure 6.5(b), ColorSafe computes $LocR_i \cap LocR_j \cap RemW_k$, for all i and j, where $i \neq j$, and for values of k that depend on whether ColorSafe is in debugging mode or deployment mode, which we discuss shortly. If the resulting signature is not empty, then it is likely that the execution contains an unserializable interleaving involving the color(s) in the resulting set. Testing for the remaining cases in Table 6.2 is analogous, except signatures of the applicable type are intersected.

Two issues remain: (1) determining when to evaluate the detection expression and (2) choosing values of k. These choices depend on whether ColorSafe is in debugging mode or in deployment mode.

Debugging mode. In debugging mode, we want ColorSafe to detect only unserializable interleavings that actually occur. We also want to know the instruction address of the memory operation that led to the interleaving. Therefore we choose k such that $i \leq k$ and we use a set containing only the most recent memory operation issued locally as the second local set in the intersection (instead of all possible H_j). This implies that only history items that actually interleave history item H_i and the most recent memory operation are considered. Figure 6.6(a) illustrates this process. If the resulting set is not empty, then an unserializable interleaving involving the just-issued memory instruction is reported. This includes both the instruction address and the type of interleaving (cases in Table 6.2). Note that for efficiency, the set of all intersections need not be evaluated from scratch at every memory operation, because the partial intersection between local and remote history items can be reused until the corresponding history items are pushed out of the history buffer.

Deployment mode. In deployment mode, we want ColorSafe to detect *potentially* unserializable interleavings. Hence, ColorSafe performs 3-way intersections consisting of all pairs

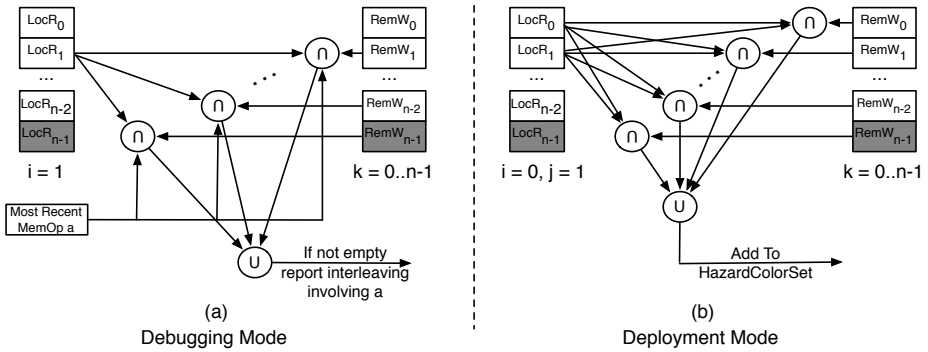


Figure 6.6: **Detecting unserializable interleavings in (a) debugging mode and (b) deployment mode.** In (a), only actual interleavings are being considered for the serializability test: the current access to a, the local history item i and the remote history items with $k \geq i$. In (b), all items in the remote history are being considered for the serializability test: local history item i, followed by local history item j, and all possible remote history items ($k = 0...n-1$).

of distinct local signatures and each remote signature, regardless of whether the history items actually interleave (*i.e.,* $k = 0...n-1$), shown in Figure 6.6(b). When the result set is not empty, it is added to the HazardColorSet. Also, we do not need to know the instruction that led to a potential atomicity violation — we need only the colors to enable dynamic avoidance. This allows us to perform detection only at the end of an epoch instead of at each memory access, which significantly decreases the frequency of intersection operations.

6.3.4 Support for Dynamic Avoidance of Multi-variable Atomicity Violations

Whenever a potentially unserializable interleaving happens in deployment mode, ColorSafe adds the color involved to the HazardColorSet. Each ColorSafe processor has its own HazardColorSet. This set is also encoded as a signature. Insertions are a bit-wise *OR* operation between the detection intersections' results and the HazardColorSet itself. Upon a memory access, the processor checks whether the access' ColorID is in the HazardColorSet. If so, it starts an ephemeral transaction to prevent a potentially unserializable interleaving. If there is an ephemeral transaction already in progress, the event is ignored, *i.e.*, transactions do not nest.

Implementing Ephemeral Transactions. Ephemeral transactions can be implemented as typical memory transactions [3, 1]. Unlike regular transactions, though, ephemeral trans-

actions are implicit, as they do not rely on code markers. As such, they do not guarantee that a set of dynamic instructions will always execute atomically and in isolation. An ephemeral transactions implementation must provide strong atomicity, since they must roll back in the event of conflicts with any remote accesses and attempt to execute again. To guarantee forward progress, an ephemeral transactions implementation can must recognize repeated rollbacks, as in prior work [27]. On detecting repeated rollbacks, ColorSafe can then reduces the size of the ephemeral transaction until it is able to commit or falls back to non-transactional execution.

6.3.5 Discussion on Hardware Complexity

Although ColorSafe requires additional hardware support, its cost is reasonable and leverages well understood technology. Mechanisms to keep track of sets of addresses and memory tagging have been proposed before (*e.g.,* Mondrian Memory Protection [133] and Loki [142], IBM 801 [29]). Mondrian and Loki use hierarchical data structures to map memory regions to tags, keeping storage overheads manageable. The buffers and logic required to handle history items are very simple, since they are based on address signatures [28, 117]. The additional support in the coherence protocol involves only an extra field in request messages and does not change any protocol state machine. Finally, support for transactional memory is being considered for actual off-the-shelf processors [3, 1].

6.4 Debugging with ColorSafe

We have developed a debugging methodology that accompanies ColorSafe's debugging mode. The program counters where ephemeral transaction are inserted indicate points where potential atomicity violations occurred. We report the set of such program counters for programmers to examine.

We refine the set of program counters reported using an invariant-based technique. The key idea is to focus on program counters reported consistently in a set of program executions to reduce the rate of false positives.

Invariant-based reduction of false positives. ColorSafe's invariant mechanism requires developers to run the program multiple times and classify each execution as buggy or non-

buggy by according to whether a failure occurred. For each execution, ColorSafe reports the set of program counters where unserializable interleavings were detected and the type of interleaving (Table 6.2). We call each pair made up of a program counter and an interleaving type a *detection identifier*. The set of detection identifiers from buggy runs are added to the *buggySet*, and those from non-buggy execution are added to *nonBuggySet*. We then set-subtract *nonBuggySet* from *buggySet*, producing the *vioSet*. This set contains the detection identifiers for unserializable interleavings that occurred only during the buggy runs. These are the points in the code on which a developer should focus to locate the bug. In Section 6.5.4, we show that this simple technique actually prunes most false positives.

6.5 Evaluating ColorSafe

Our goals in evaluating ColorSafe are to assess how well deployment mode dynamically avoids atomicity violations (Section 6.5.2) and at what performance cost (Section 6.5.2), to understand design trade-offs (Section 6.5.2 and Section 6.5.2), and to assess how accurately debugging mode locates bugs in the code (Section 6.5.4).

6.5.1 Experimental Setup

We evaluated ColorSafe in a simulator built using the Pin [87] binary instrumentation framework [87]. The simulator models all ColorSafe structures, including the Multilevel Color Table, translation of data addresses to colors, the history buffer, unserializable interleaving detection using signature operations on history items, the HazardColorSet, and ET support. The simulator models both debugging and deployment mode. In debugging mode, it produces the unserializable interleaving detection output that is used by our invariant-based debugging framework. In deployment mode, the simulator determines how often atomicity violations were avoided by determining whether the violation executed entirely within an ET. To assess performance impact, we model ET conflicts.

We use a variety of benchmarks consisting of "bug kernels" and full applications. Table 6.1 provides a description of each kernel and application, along with the portion of the dynamic execution spent in buggy code (Column 4) and the interleaving pattern that causes the bug (Column 5). The bug kernels are segments of buggy code extracted from

full applications. We extracted five kernels from various versions of the Mozilla Project, all previously discussed in the literature [78, 79]. We paid special attention to maintaining the original data structure hierarchies and the layout of the code surrounding the bug. As our full applications workloads, we use the AGet parallel download accelerator, the Apache httpd webserver and the MySQL database server. To exercise the buggy regions of Apache, we used scripts to repeatedly launch 100 concurrent requests. We exercised the MySQL bug using a version of the sql-bench benchmark modified to execute many concurrent requests. The bug in AGet involves a signal handler, so to exercise the buggy code, we fetched a file from a network resource, and interrupted the transmission with a Unix signal.

	Name	**Description**	**% Bug Ex.**	**Intlv. Type**
Kernel	nsText	Mozilla-0.9: During update of string buffer offset and length, inconsistent data can be read.	0.19%	WRW
	NetIO	Mozilla-0.9: Read of flag and conditional write can be interleaved, invalidating data.	0.14%%	RWW
	jsStr	Mozilla-0.9: Between update to string buffer and length, inconsistent data can be read by remote read.	0.22%	WRW
	interp	Mozilla-0.8: Between table update and flag update interleaving can make table inconsistent.	2.8%	WWW
	msgPane	Mozilla-0.8: Interleaving read of flag indicates content loaded in msg. pane before content is loaded.	0.22%	WRW
Full	Ap2.0	Apache-2.0.48: Character buffer and string length made inconsistent by concurrent accesses.	0.91%	WRW
	AGet	AGet-0.4: During update of log contents/length, inconsistent data can be read by signal handler.	0.47%	WRW
	MySQL	MySQL-3.23.57: Accesses can be logged out of order by highly concurrent access to replay log.	33.21%	WWW

Table 6.1: **Bugs used to evaluate ColorSafe.**

We experiment with both manual coloring and malloc-coloring, as described in Section 6.2.1. To perform manual coloring, we added explicit annotations to the code to associate colors with data. For malloc-coloring, our simulator monitors calls to memory allocation functions and assigns a new color to the allocated region.

6.5.2 Deployment Mode: Bug Avoidance

We start by showing that ColorSafe is able to avoid most atomicity violations in bug kernels and applications. All experiments had epochs of 400 instructions, a total history window of 12,000 instructions (*i.e.,* 30 history items), and 3,000-instruction ETs. Table 6.7 shows the number of violation instances avoided.

	Benchmark	% Avoided *Manual*	% Avoided *Malloc*
Kernel	nsText	99.95	99.95
	NetIO	99.95	99.95
	jsStr	100	100
	interp	99.95	0
	MsgPane	99.95	0
Full App.	Ap2.0	98.72	94.18
	AGet	99.28	0
	MySQL †	77.0	71.4

Figure 6.7: **Violations avoided in bug kernels and full applications.** Results are shown for experiments using manual and malloc data coloring. †We used a different system configuration for MySQL. We explain the details in Section 6.5.2 (Difficulties with MySQL).

App.	% ET Start	% in ETs	% Usfl ET	% in Usfl ET	% Usls Cflct	% in Usls ET w/ Cflct
Ap2.0	0.02	38.4	7.4	5.8	4.1	3.2
AGet	0.005	12.8	63.8	10.7	6.4	1.1
MySQL	0.003	24.7	9.0	20.2	0.5	1.2

Table 6.2: **Characterization of Ephemeral Transactions.** The rate of ET starts, % of useful ETs, and % of conflicting useless ETs for full applications in deployment mode. Ap2.0 and MySQL were run using malloc coloring, and AGet, manual coloring. MySQL was run with the modified configuration described in Section 6.5.2 (Difficulties with MySQL).

For bug kernels, ColorSafe avoids nearly 100% of the violation instances using manual coloring. Malloc-coloring is capable of avoiding almost all atomicity violations in most kernels, but it is not effective for all kernels. The bugs in *interp* and *msgPane* each involve accesses to one global variable, and one dynamically allocated variable. They are not allocated together, so using malloc-coloring does not capture their correlation. As a result ColorSafe is unable to avoid these bugs.

Table 6.7 shows that ColorSafe avoids nearly all atomicity violations in our full application benchmarks. In runs of Ap2.0, ColorSafe avoids virtually all instances of the violation, using both manual and malloc-coloring. Malloc-coloring has a slightly lower rate of avoidance. This is because ETs triggered by accesses to data unrelated to the violation end up preventing useful ETs from proceeding. In runs of AGet, ColorSafe avoids more than 99% of instances of the violation using manual coloring. The bug in AGet involves a dynamically allocated variable and a global variable, so unfortunately malloc-coloring is unable to identify their correlation.

Difficulties with MySQL. ColorSafe was unable to avoid violation instances in MySQL using our standard configuration. This is because the violation is nearly 20,000 instructions long, and cannot execute entirely within an ET of 3,000 instructions. We re-ran MySQL using 64,000-instruction ETs, 1,000-instruction epochs, and a total history window of 30,000

instructions. With this configuration, ColorSafe avoids 77% of the violations using manual coloring, and 71.4% of violations using malloc-coloring. ColorSafe's less-than-perfect avoidance using even this configuration results from longer ETs triggered in response to false positive detections that prevent useful ETs from beginning over a much longer window.

Could Avoidance Happen by Chance? A Comparison with Random Ephemeral Transactions. One may wonder whether the bug avoidance achieved by ColorSafe would be possible simply by starting ETs at random points. Here we show empirically that this is not the case. Consider an experiment using AGet. Using 3,000-instruction ETs and at random starting about 5 per 100,000 dynamic instructions (the rate of transaction starts for AGet using standard ColorSafe) avoids 1.8% of all violations. ColorSafe is able to avoid 99.28% of all violations using the same configuration. Performing the same experiment with Ap2.0, we see that random ETs avoid only 6.97% of violations. ColorSafe avoids 98.72% of violations. Results were similar for other benchmarks. This stark contrast shows empirically that ColorSafe avoids significantly more potential atomicity violations than chance.

Performance Overheads

ColorSafe in deployment mode imposes modest impact on performance. We now discuss and quantify the key sources of overheads, which are coloring support and ETs. ColorSafe leverages existing cache coherence support to handle the exchange of color information between processors. As a result, communicating color information imposes negligible run time overhead. Color information lookup depends on the meta-data scheme underlying ColorSafe. While a lookup is not free, the cost is minimized using caches for meta-data information. Moreover, color information is mostly read-only (*i.e.,* written only at allocation time). This means that any additional overhead associated with meta-data writes is unlikely to affect performance.

The main sources of performance degradation in ColorSafe are the bookkeeping overhead of ETs and the cost of re-executing ETs due to conflicts. In Table 6.2, we report the percentage of dynamic instructions that triggered an ET (% ET Start), the number of ETs that were useful in preventing an atomicity violation (% Usfl ETs), and the percentage of

useless ETs that experienced a conflict (% Usls Cflct). We report the number of ET starts as a fraction of the total number of dynamic instructions to quantify how often the overhead of starting an ET is incurred. The fraction of useful ETs is a measure of how often the cost of an ET was worthwhile, because it prevented a violation. The fraction of conflicting useless ETs quantifies the amount of work wasted in ETs that did not avoid an atomicity violation and still had to be re-executed. In Table 6.2, we also show the fraction of total dynamic instructions that executed inside ETs (% in ETs), the fraction that executed in useful ETs (% in ET Usfl), and the fraction that executed in useless ETs that had conflicts (% in Usls ET w/ Cflct). We report data only for full applications, as kernels execute in tight loops around buggy code, making them unsuitable for this analysis.

There are two important results in these data. First, for all applications, the rate at which ETs are triggered is very low: 3 ETs per 100,000 instructions for MySQL, 5 per 100,000 for AGet, and 20 per 100,000 for Ap2.0. The low frequency of ET starts indicates that the cost of starting, ending and verifying ETs will have little effect on performance. We also see a relatively small fraction (12–38%) of the execution is executed transactionally.

Second, very little computation is wasted by re-executing useless ETs. The data show that the fraction of useful ETs ranges from 7.4% (Ap2.0) to 63.8% (AGet). At first glance this may suggest that useless ETs are frequent, and hence problematic. However, the rate of aborts for useless ETs is very low — just 0.5% for MySQL, and at most 6.4% in AGet. The work wasted in these useless, aborted ETs amounts to just a small fraction of dynamic instructions, from 1.1% to 3.2%. Thus, if an ET is useless, it rarely experiences a conflict, and very little work is wasted. If an ET is useful, it is more likely to abort, but we consider it profitable to sacrifice this small amount of performance in exchange for prevention of buggy behavior. Additionally, only a small fraction of the execution (5.8%–20.2%) executes in useful ETs and incur the higher likelihood of abort.

Sensitivity to Ephemeral Transaction Length

Table 6.3 shows avoidance for each application as the size of ETs is varied between 3,000 and 15,000. For all the bugs we considered (except the very long MySQL bug), ColorSafe's

avoidance is stable for the ET sizes shown. This insensitivity to ET size shows two things: (1) large ETs do not inhibit the avoidance capability of ColorSafe; and (2) there is flexibility in the selection of this design parameter. We chose a default ET size of 3,000 instructions; any smaller, and we risk being unable to avoid modestly large violations; any larger and we increase the chances of unnecessary abort.

App.	% Violations Avoided			
	3,000 Inst ET	***5,000 Inst ET***	***10,000 Inst ET***	***15,000 Inst ET***
nsText	99.95	99.95	99.95	99.95
NetIO	99.95	99.95	99.95	99.95
jsStr	100.0	100.0	100.0	100.0
interp^m	99.95	99.95	99.95	99.95
msgPane^m	99.95	99.90	99.90	99.95
AGet^m	99.28	97.93	99.10	99.18
Ap2.0	94.18	90.55	98.64	94.16

Table 6.3: **Failure avoidance for a variety of ET sizes.** Applications marked with a ^{m} were run using manual coloring, because their bugs involve global and heap variables; All others were run with malloc-coloring.

Sensitivity to History Buffer Configuration

The history buffer configuration determines which interleavings are observable by ColorSafe and affects which unserializable interleavings can be detected. We now evaluate the effect of varying the history buffer configuration. We do this by injecting "noise" into a bug kernel to simulate high-frequency concurrent access to shared data. We added noise by allocating additional data unrelated to the bug in the kernel. Randomly, 1% of the time, these data were given the same color as the data involved in the bug. During execution, five extra threads, in addition to the threads originally in the benchmark, repeatedly access the "noise data". They make a random number of accesses between 1 and 10 and determine whether to read or write by "flipping a coin". The rate at which these additional threads access data is the "noise level" of the experiments.

Figure 6.8 shows avoidance in the presence of noise, with a fixed total instruction history (12,000 instr.) and using both coarse (1,200 instr.) and fine (400 instr.) history items. These data show that as the noise level decreases, avoidance improves. We see the improvement because as noise decreases, the number of ETs triggered by accesses unrelated to the bug decreases, permitting useful ETs to proceed (recall ETs don't nest).

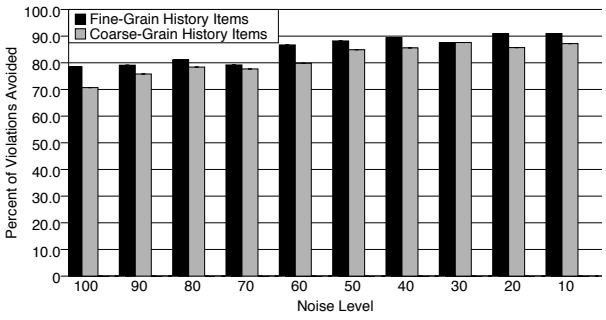


Figure 6.8: **Impact of history item granularity on violation avoidance.** The plot shows the number of atomicity violations avoided in kernel NetIO under synthetic noise for fine- and coarse-grain history items with a constant history window.

Amt. Noise	% Usf. ETs	
	Fine	*Coarse*
100	6.47	5.35
90	7.05	6.19
80	8.06	6.65
70	8.40	7.04
60	9.85	7.83
50	10.88	8.91
40	11.72	9.61
30	11.90	10.62
20	13.36	10.26
10	13.11	11.71

Table 6.4: **Impact of noise on ET usefulness.** Percentage of useful ETs in NetIO with synthetic noise, 12,000-instruction total history length, and varied history item granularity.

This figure also shows that using fine- rather than coarse-grain history items improves avoidance. The reason is that fine-grain history items encode the relative order of memory accesses more precisely. Using coarse history items, more accesses occur within the same history item and are considered simultaneous. This prevents ColorSafe from seeing an interleaving of these accesses. When fine-grain history items are used, ColorSafe is able to observe these interleavings, enabling earlier detection of buggy interleavings.

Table 6.4 shows the proportion of useful ETs in these experiments. There is an inverse relationship between the noise level and the fraction of useful ETs. This relationship corroborates our above conclusion: avoidance is impeded in the presence of noise because more ETs are useless. The data also show that using fine-grain history items results in a larger fraction of useful ETs. This is in agreement with our findings from Figure 6.8, that fine-grain history items lead to more precise detection.

6.5.3 Characterizing Meta-Data Requirements

The size of the color meta-data in ColorSafe is derived from the total number of necessary colors. In our experiments using malloc-coloring, we saw as few as 265 colors in Ap2.0, around 900 colors in MySQL, and as many as about 4,015 colors (in jsStr). Using manual coloring, we saw just a single color in most cases (kernels, AGet, MySQL) and at most

5 colors in Ap2.0. These numbers represent the total number of coloring events (*e.g.*, allocations) in the execution.

We chose to use 12 bits of meta-data to represent colors in ColorSafe. For manual coloring, 12 bits provides ample space, according to our experimental results. Using malloc-coloring, capacity will be an issue for applications that frequently allocate memory. One way of handling this is to recycle colors when their associated memory is deallocated. Also, at additional cost, ColorSafe could be implemented with wider meta-data fields.

6.5.4 Debugging Mode: Locating Bugs in the Code

We focused on full applications for our debugging experiments. We collected the report of unserializable interleavings generated by ColorSafe using both manual coloring and malloc-coloring (when possible). We present a comparison of the detection capability in deployment mode versus debugging mode to show the effect of a stricter detection policy. We quantify the report in terms of code points — *i.e.*, lines of code to inspect.

Using malloc-coloring in debugging mode, ColorSafe reported a large number of detection code points — 1,493 for Ap2.0. By applying invariant-based processing, we saw a marked reduction in detections to just 58. This much smaller set of detections would lead a programmer directly to a bug, or short of that, would help a programmer decide how to manually color data.

We foresee ColorSafe being more useful for debugging if data structures that are suspected to be related to a bug are manually colored by a developer. We consider it reasonable to assume a programmer would be able to do this, given a standard bug report, output from ColorSafe using malloc-coloring, and knowledge about the data structures in the program. Table 6.5 shows the number of code points produced by ColorSafe using manual coloring. We show results for deployment mode (Column 2), debugging mode (Column 3), and using our invariant approach described in Section 6.4 (Column 4). ColorSafe is able to detect the bug in Ap2.0, with only 2 false positives (3 code points) in deployment mode, and just 1 false positive (2 code points) in debugging mode. ColorSafe detects the bug in AGet with just a handful of false positives as well, in both deployment mode, and debugging mode.

Two other facts stand out among these data. First, there is a decrease in detections from deployment mode to debugging mode. This is because debugging mode has a stricter policy for determining that an interleaving is unserializable (Section 6.2.2). There is a reduction of approximately 21% (256 code points) in the number of code points reported for MySQL, and 17% (4 code points) for AGet. However, further improvement is still desirable — even necessary — with hundreds of code points left to sift through for MySQL. This brings us to our second result: The reduction in reports resulting from applying invariant-based processing is dramatic. Invariant-based pruning eliminates hundreds of false positives for MySQL, leaving 40 code points to be analyzed in a software package of over a million lines. For AGet, we reduce the number of code points reported to just 8.

This shows that ColorSafe debugging, coupled with manual coloring and our invariant-based approach, leads to very few false positives. The reason that the invariant based approach works so well is that interleavings that occur in non-buggy executions and those that occur in buggy runs tend to be very similar. These similarities are filtered out by our invariant-based approach, leaving only relevant detections. In addition, the nondeterminism in multiprocessor systems provides diversity of behavior when we execute the application multiple times. It is realistic to assume that the programmer can manually color data because it requires only local reasoning, when the data is declared (or allocated). Reasoning of this sort is the basis of prior proposals focusing on data-centric synchronization models [26, 57, 126].

Benchmark	**# Detections**		
	Deployment	***Debugging***	***Post-Processed***
Ap2.0	3	2	**2**
AGet	24	20	**8**
MySQL	821	677	**40**

Table 6.5: **Evaluation of ColorSafe for debugging.** Number of code points reported by ColorSafe using deployment mode, debugging mode, and debugging mode with invariant post-processing.

6.6 Conclusions, Insights, and Opportunities

This chapter introduced ColorSafe, an architecture that simplifies concurrency bug debugging and dynamically avoids schedule-dependent failures. A major contribution of ColorSafe

is its ability to handle both single- and multi-variable atomicity violations. The key idea is to group semantically related variables into colors and to perform serializability analysis in the "color space". Color-space serializability analysis enables detection of likely atomicity violations involving sets of variables, not just a single variable.

ColorSafe has two modes of operation: debugging mode, which produces detailed information about how and where atomicity violations may have happened and deployment mode, which is less precise, but uses transactional memory support to avoid failures without affecting the semantics of the program. Our results show that ColorSafe effectively avoids atomicity violations in both bug kernel benchmarks and several real applications, including Apache and MySQL. ColorSafe is efficient and avoids atomicity violations with little performance overhead. ColorSafe's debugging mode precisely identifies code related to atomicity violations with few false positives, using a simple invariant-based technique to prune spurious reports.

Insights. There are several major insights that we gained from this work. We showed that there is synergy between existing techniques for failure avoidance [85, 27, 56, 32], data-centric synchronization [26], and atomicity debugging [80, 85]. We also showed that instruction interleaving behavior in failing and non-failing program executions is largely similar. The similarity was essential to ColorSafe's debugging methodology. We also showed that hardware transactional memory support is adequate to avoid atomicity violations – transactions-all-the-time, like Atom-Aid used (Chapter 5) is not required for effective failure avoidance.

Opportunities. There are several interesting avenues for future work stemming from ColorSafe. One interesting direction is to determine whether hardware support is fundamentally required for the style of failure avoidance used by Atom-Aid and ColorSafe. There are two ways this style of failure avoidance could be realized without hardware support. One way is to build a runtime system that dynamically identifies potential atomicity violations (like ColorSafe's hardware mechanism does) and to use a software transactional memory implementation to enforce the required atomicity property. Another way would be to distill out likely atomicity requirements and modify program code to enforce those constraints using synchronization like locks, or explicitly expressed transactions.

The main limitation of both Atom-Aid and ColorSafe is that they do not deal with concurrency bugs in general. Atom-Aid and ColorSafe address failures due to atomicity violations only. Chapter 7 addresses this shortcoming with a failure avoidance mechanism that is effective for a much more general class of concurrent program failures.

Chapter 7

AVISO: AVOIDING SCHEDULE-DEPENDENT FAILURES

This chapter describes a technique for avoiding general schedule-dependent failures, without relying on bug-specific heuristics. Chapters 5 and 6 described Atom-Aid and ColorSafe, two techniques that also focused on avoiding schedule-dependent failures, due to atomicity violations. Atomicity violations are an important and widespread concurrent program failure mode. Focusing on atomicity violations allows these techniques to take advantage of *serializability*, a property that is uniquely helpful for detecting and avoiding problems related to atomicity. The drawback of focusing on such a specialized property is that these techniques are not *generally* applicable to arbitrary concurrent program failures.

Designing a general schedule-dependent failure avoidance mechanism presents many interesting challenges. Such a system must monitor program execution to collect the information needed to identify likely failures. Monitoring must be efficient because failure avoidance is most useful in deployed systems, where performance is paramount. Such a system also requires precise techniques to identify failures using the collected data and mechanisms to influence the program's execution to avoid failures; as with monitoring, such mechanisms must be efficient. Furthermore, systems should avoid failures without requiring modifications to hardware or changes to the way programmers write their programs.

We overcome these challenges in this chapter. Our main contribution is a *novel, automated technique for avoiding schedule-dependent failures*. We develop an efficient system for collecting relevant program events at run-time in deployed software. When a program instance fails, we use the information collected by our system to generate hypotheses about what caused the failure. Then, leveraging the fact that we often have a number of deployed instances of the same software, we develop a predictive statistical model and an empirical framework to identify which hypothesis is most likely to be correct. Based on that hypoth-

esis, we influence future program executions by perturbing the thread schedule to avoid subsequent failures.

We implement these ideas in Aviso, a distributed system that coordinates many deployed instances of a program to cooperatively learn how to prevent failures. Our system works on commodity hardware with minimal impact on software and the development process. Our evaluation on several real-world desktop, server, and cloud applications shows that Aviso effectively and efficiently avoids schedule-dependent failures. In one test, Aviso reduced the failure rate of a buggy version of Memcached, a popular key-value store, by two orders of magnitude. Fixing the bug took developers about a year and their fix imposed about 7% performance overhead [36]. Aviso worked without developer intervention and in minutes found an effective constraint that imposed only a modestly higher overhead of about 15%.

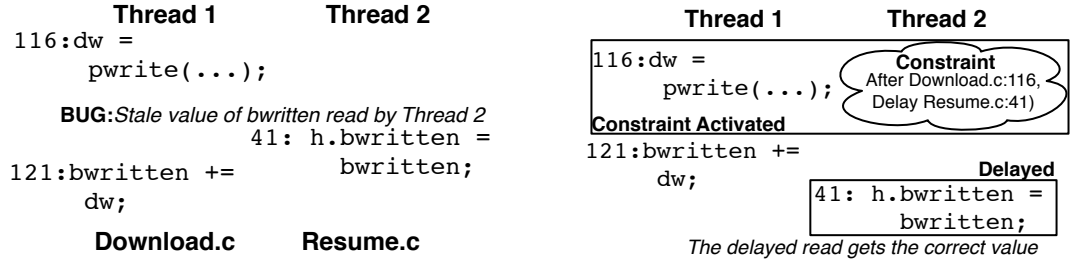
The remainder of this chapter describes Aviso in greater detail. Section 7.1 provides background that explains how Aviso avoids schedule-dependent failures. Section 7.2 reviews Aviso's design goals and architecture, while Section 7.3 relates how Aviso decides which events to monitor during an execution and how events and failures are to be monitored. We describe how Aviso generates candidate constraints after a failure in Section 7.4 and how it determines which candidate constraint effectively prevents failures as well as inter-system constraint sharing in Section 7.5. Section 7.6 provides implementation details. Sections 7.7–7.8 evaluate and characterize Aviso, contrast it with prior work, and summarize our findings.

7.1 Schedule-Dependent Failures

A program's execution experiences a schedule-dependent failure when threads execute a particular sequence of events that leads to a crash, contract violation (*e.g.*, assertion), or state corruption. Schedule-dependent failures are the result of programming errors, such as absent or incorrectly used synchronization.

Figure 7.1(a) illustrates how a programming error leads to a failure using an example from AGet-0.4, a multi-threaded download accelerator. Figure 7.1(a) shows a snippet of a failing multi-threaded execution. The failure is an atomicity violation: Thread 1 writes bytes to a file (`pwrite(...)` on line 116) and then adds the number of bytes written to the file to a shared counter (incrementing `bwritten` on line 121). The failure occurs

when Thread 2 asynchronously reads the value of `bwritten` on line 41, between the call to `pwrite()` and the increment that follows. The programmer has omitted synchronization operations in Thread 2, permitting that thread to read an intermediate value. Note that if Thread 2's read was delayed, and Thread 1's update was allowed to proceed, the failure could have been avoided.



(a) **A failing execution.** The failure occurs when Thread 2 reads a stale value of `bwritten`.

(b) **Avoiding the failure.** The constraint delays Thread 2, reordering operations to avoid the failure.

Figure 7.1: **A schedule-dependent failure in AGet-0.4.**

7.1.1 Bug Depth

Recent concurrency testing work [25] formally characterized concurrency errors using the notion of *bug depth*. A bug's depth is the minimum number of pairs of program events that must occur in a particular order for that bug to manifest a failure [25]. Note that a single *bug* can lead to different *failures* under different execution schedules. A different set of event pair orderings is necessary to manifest each different failure and a failure occurs only if all its necessary orderings are satisfied. Typically, most schedules do not satisfy all orderings required to cause any particular failure, so executions usually do not fail.

This fact poses a key challenge to multi-threaded testing. To expose a bug during testing by causing a failure, systems must enforce a number of orderings on the execution schedule that is greater than or equal to the bug's depth. The larger the bug's depth, the more work is needed to ensure that a failure occurs [25]. The bug in Figure 7.1(a) has depth 2 because two pairs of events must execute in a particular order for the failure to manifest: Thread

2's read must follow Thread 1's `pwrite` call and must precede Thread 1's `bwritten` update. If the first ordering were not upheld, Thread 2 would correctly see the value of `bwritten` before any of Thread 1's operations. If the second ordering were not upheld, Thread 2 would correctly see the result of Thread 1's operations.

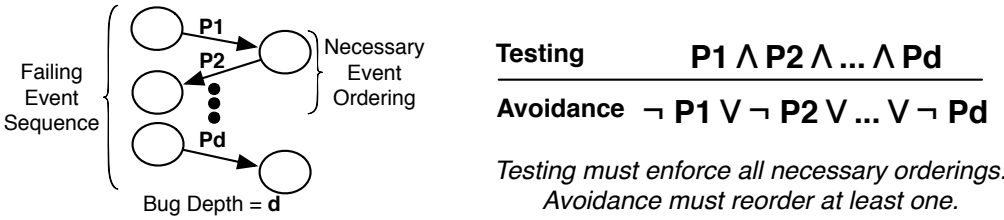
7.1.2 The Avoidance-Testing Duality

In this work, we make the observation that a failure caused by a bug of *any* depth can be avoided by perturbing just *one* pair of events in the sequence that leads to the failure. Given a chain of pairs of events that must be ordered to manifest a failure in testing, perturbing the schedule to reorder any of the pairs "breaks the chain", preventing that failing schedule from occurring. This observation suggests a duality between testing for schedule-dependent failures and avoiding them: testing requires enforcing *all* of a conjunction of pair orderings to exercise a failing schedule; avoidance requires reordering events in *any* of those pair orderings to avoid a failing schedule. Avoiding a particular failing schedule leads the execution to a new schedule. Avoidance is successful if the new schedule does not lead to a failure, which is likely the case since there are typically significantly more failure-free schedules than schedules with failures.

Figure 7.2(a) illustrates bug depth. The circles are program events and the arrows represent pair orderings necessary to lead to a failure. There are d orderings, so assuming the figure shows the fewest possible orderings for the bug to cause a failure, the bug has depth d. Figure 7.2(b) contrasts testing and failure avoidance. Exposing a bug during testing requires satisfying a conjunction of d pair orderings, whereas the failure is avoided if a single pair of events is reordered.

7.2 System Overview

This section provides an overview of Aviso's system architecture and design constraints. We then walk through Aviso's failure avoidance mechanism with an example, and explain how Aviso facilitates *cooperative*, *empirical* failure avoidance in a community of software instances.



(a) **Pair orderings necessary for the failure to occur**. There are d orderings, so the bug responsible for the failure has depth d (assuming the figure shows the fewest orderings for the bug to cause a failure).

(b) **Contrasting testing with failure avoidance.** Testing aims to expose bugs, so *all* orderings must be satisfied. Avoidance aims to prevent failures by reordering the events in *any* ordering.

Figure 7.2: **The Avoidance-Testing Duality.**

7.2.1 System Architecture

Figure 7.3 presents Aviso's four components: (1) the profiler, (2) the compiler, (3) the runtime system, and (4) the framework.

Profiler and Compiler. After development, the programmer runs Aviso's profiler, which determines what program operations to monitor. The profiler sends an event profile to Aviso's compiler. The compiler uses the event profile to add event-handling runtime calls to the binary and links the binary to the Aviso runtime to produce an Aviso-enabled executable.

Runtime. The Aviso runtime system monitors and keeps a short history of events during program execution. The runtime also watches for failures and alerts the framework when they occur. Periodically during execution and when execution fails, the runtime sends its event history to the framework. The runtime can also perturb the execution schedule using *schedule constraints* (see below) that it receives from the framework.

Framework. Aviso-enabled executables run in the Aviso framework and the two communicate via a simple messaging API. The framework collects event histories and failure information sent by the runtime. It generates schedule constraints from this information and sends them to the runtime when Aviso-enabled executables start running. The framework selects constraints to send using a statistical model that predicts which constraints

are most likely to avoid failures; it builds the model using aggregated history and failure information. By aggregating information from and sending constraints to many program instances, the framework enables program instances to cooperatively avoid failures.

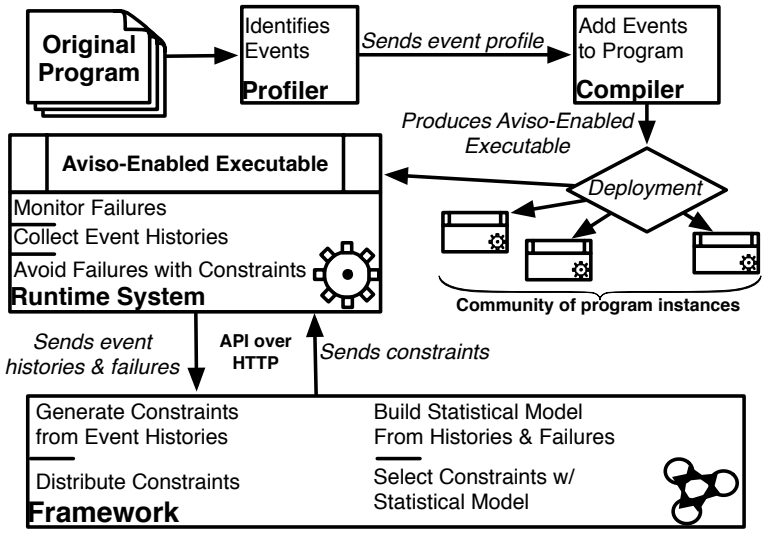


Figure 7.3: **Aviso's components.** The compiler and profiler find and instrument events. The runtime system monitors events and failures and avoids events. The framework generates constraints, selects likely effective constraints using a statistical model, and shares effective constraints in a community of software instances.

Design Requirements

Our goal is to build a system that is general enough to avoid failures due to a broad class of concurrency errors in a deployment setting. This goal presents us with conflicting design constraints: For the sake of generality, the runtime should monitor as many program operations as possible, to capture a large variety of failure behaviors. However, monitoring imposes a time and space overhead, and building a system intended for use in deployment necessitates high performance. Aviso should find effective constraints quickly and permit as few failures as possible. To do so, Aviso must leverage *prediction* to identify effective constraints without having to directly observe their impact on the program's behavior.

Aviso is additionally constrained because programmer time is valuable and understanding and fixing concurrency errors is difficult, error-prone, and time consuming. Our system should require as little as possible from the programmer.

7.2.2 Avoiding Failures with Schedule Constraints

Aviso leverages the avoidance-testing duality introduced in Section 7.1.2 to avoid failures. Aviso uses *schedule constraints* to perturb an execution's thread schedule with the goal of reordering events in the execution whose original order leads to a failure.

A constraint is based on a pair of events observed by Aviso in a failing program execution. It is "activated" when the first event in its pair executes. When a constraint is active and a thread tries to execute the second event in the constraint's pair, Aviso delays the thread, reordering some events in the execution. If the original order of the reordered events was necessary for a failure to occur, reordering the events will avoid the failure. Such reorderings are the key to Aviso's failure avoidance mechanism. We provide more details on avoidance in Section 7.4.2.

Figure 7.1(b) shows how Aviso uses a constraint to avoid a failure. The constraint is made from a pair of events: the first event is Thread 1's `pwrite` call at Download.c, line 116, and the second event is Thread 2's read of `bwritten` at Resume.c, line 41. When the first event is executed, the constraint is activated. While the constraint is active, Thread 2 attempts to execute the second event, and it is delayed for a fixed period. During this delay, Thread 1's `bwritten` update executes atomically with its `pwrite` call, preventing the failure. Later, when Thread 2 resumes execution, it reads the correct value of `bwritten`.

7.2.3 Cooperative Empirical Failure Avoidance

Aviso is an *empirical* failure avoidance system. When a failure occurs, it generates a set of constraints from its event history (see Section 7.4). Each constraint is a *hypothesis* about how to prevent the failure. Aviso decides which constraints are most likely to avoid failures and instructs program instances to use those constraints. It is an *empirical system* because it uses a combination of predictive statistical modeling and experimentation to select effective

constraints. As Aviso observes more execution behavior, it refines its model, improving its selections. The details of constraint selection are described in Section 7.5.

Aviso is also a *cooperative* failure avoidance system. It leverages communities of computers running the same program to select effective constraints in two ways. First, Aviso's statistical model is built using information drawn from a community of program instances. Second, Aviso distributes constraints to all members of a community. A constraint that consistently avoids failures for some members can be distributed to other members, sharing its benefit.

7.3 Monitoring Events and Failures

Aviso's profiler identifies program operations relevant to concurrent program behavior. The compiler inserts event-handling runtime calls into Aviso-enabled executables before each operation. Aviso works on deployed programs, so determining which operations should be treated as events determines the overhead of event handling. Aviso uses static and dynamic analyses to prune the detected set of events. Aviso's runtime traces events during execution and monitors for failures.

7.3.1 Identifying Relevant Program Events

Aviso focuses on concurrency errors, so we restrict our attention to events related to concurrency. There are three types of events that Aviso monitors: (1) synchronization events, (2) signal events, and (3) sharing events. *Synchronization events* are lock and unlock operations, thread spawn, and join operations that can be identified by matching synchronization library (*e.g.*, `pthread`) calls. Aviso can handle other types of synchronization, as well (*e.g.*, CAS) if the programmer identifies them as synchronization.

Signal events are functions that handle signals. They are of interest because signals may be delivered and handled asynchronously. Signal events are identified by instrumenting signal handler registration functions (*e.g.*, `signal()`) and functions that explicitly wait for signal delivery (*e.g.*, `sigwait()`).

Sharing events are more difficult to identify because they cannot be identified by looking only at syntactic properties of a program. Instead, Aviso identifies sharing events using a

sharing profiler before application deployment, *i.e.*, during testing. The sharing profiler monitors threads' accesses to shared data. When a thread accesses data that has been accessed by another thread during the execution, the operation the thread is executing is considered to be a sharing event. Sharing events are reported by the profiler and inserted into the deployment binary by Aviso's instrumenting compiler. Aviso identifies events by their instruction address and a fixed-length prefix of the call stack when the event occurs.

7.3.2 Pruning and Instrumenting Events

Handling events too frequently leads to high performance overheads. To mitigate that issue, we use two techniques to reduce the number of handled events. First, our instrumenting compiler uses *dominance pruning* to eliminate instrumentation of some events. Second, our runtime system uses *online pruning* to limit the number of events that are handled.

Dominance Pruning (Static) Analysis. When compiling a function and before optimization, Aviso's compiler computes the set of dominators for each instruction. We use the computed dominance relationships to prune the set of candidate events. Given a pair of events (p, q), if p dominates q, then for every execution of q, there was a prior execution of p. Hence, tracking only p captures nearly as much information as tracking both p and q. In this situation, we remove q from the set of candidate events. If p and q are far apart in the code, dominance pruning might discard useful events. However, our analysis did not pose problems in our experiments for several reasons. First, dominance pruning does not apply to synchronization events, and synchronization events often occur near sharing events. Second, we identify sharing events using profiling, which is approximate; dominance pruning makes the approximation only slightly less precise. Third, dominance pruning operates at function granularity, limiting the distance between p and q to the length of a function at most. Fourth, if events are far apart, the dominance relation still conveys information about the interleaving of events along certain control flow paths. This information is less precise but still useful to prevent failures.

Online Pruning. To further reduce overheads, Aviso uses *online pruning* to adaptively reduce the number of events handled. During execution, Aviso tracks the interval between

consecutive events. If two events occur within 1μs of one another, they likely encode redundant information, and Aviso discards the second of the two. Discarding events is, in effect, dynamically coalescing a sequence of events that occur within 1μs of one another into a single event, represented by the first in the sequence.

7.3.3 Tracing Important Program Events

When generating schedule constraints, Aviso focuses on pairs of events that occurred in a single failing program execution. When an execution fails due to a concurrency bug, the event sequence that caused the failure must have occurred during that execution. Aviso focuses on program events that occurred just before the failure. These events are likely to be related to the failure because *some* code point must have triggered the failure (*e.g.*, caused a crash, emitted buggy output, violated a contract, *etc.*); these events must have occurred shortly before the symptom of the failure was manifested. This observation suggests that a backward scan over a trace of events from the point of failure is likely to encounter the events involved in the failure.

We therefore designed the Aviso runtime to maintain a totally ordered history of events recently executed by any thread, called the *Recent Past Buffer*, or RPB, for the execution. The RPB is a fixed-size queue; the size could vary across implementations, but it should be on the order of hundreds of events. We used an RPB that holds at most 1000 events from each thread. Across most of our experiments, we saw an average event frequency of around 500μs, so each thread's RPB covers about the last 0.5s of its execution. Half a second is likely to be long enough to capture events related to a failure, as prior work suggests that such events often occur over short windows of execution [85, 79, 80]. When an event is executed, the oldest event in the RPB is dequeued and discarded, and the newest event is enqueued. When a failure occurs, the RPB contains a history of the execution's final moments and is likely to include the events that led to the failure.

7.3.4 Monitoring Program Failures

For crashes and assertion failures, the runtime preserves the RPB before the program terminates. For other failures, Aviso monitors for *ad hoc* failure conditions and preserves the contents of the RPB when failure conditions are met.

The best way to detect non-crash failures depends on the failure's symptom. Identifying arbitrary failures automatically is a difficult problem, and doing so comprehensively is outside the scope of this work. However, simple solutions often work well; *e.g.*, validating output is often adequate. In our tests with Memcached, we added an assertion that encodes a simple data structure invariant, preventing the use of deallocated storage. Section 7.7.1 describes our experience adding failure monitors to programs for the subset of our tests that required it.

In general, given an error report that describes a failure's symptom, an *ad hoc* failure monitor can be added to the Aviso framework to handle any failure diagnosis criteria. Using failure monitors is not always necessary – Aviso deals with fail-stop errors by default. When necessary, adding failure monitors is less risky and onerous than patching code or writing a workaround [134].

7.4 Generating Constraints and Avoiding Failures

After a failure, the framework examines the RPB and enumerates event pairs that could potentially have led to the failure. For each pair, it generates a *candidate constraint* that perturbs the thread schedule around the events in the pair. A candidate is *effective* if its perturbation avoids a failure. The framework selects candidate constraints to make available to future program executions that can use them to avoid failure. In Section 7.5 we discuss how Aviso selects effective candidates.

7.4.1 Generating Candidate Constraints

We take a straightforward approach to selecting event pairs to generate constraints. The framework considers pairs of events in execution order in the RPB, (B, A), that were executed by different threads. It selects these pairs under the constraint that between B and

A no event was executed by the thread that executed B. Note that between events in a pair, other uninvolved threads may execute other unrelated events.

Figure 7.4 illustrates the process of enumerating event pairs. Notice that Thread 1's first execution of E is not part of a pair because it is immediately followed by B in the same thread. Also notice that Thread 2's F and Thread 3's X form a pair in spite of their separation by Thread 1's executions of E.

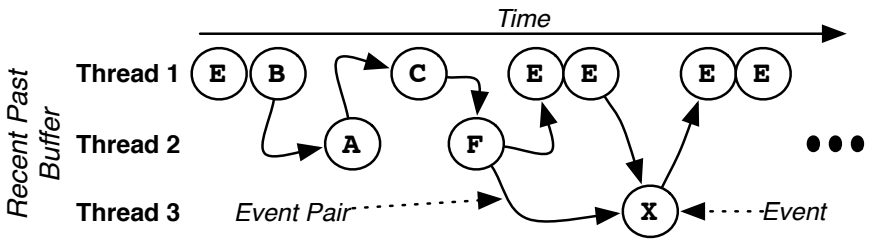


Figure 7.4: **Enumerating pairs from a failing execution's RPB.** There are three threads, and time proceeds left to right. Circles are events, and arcs between events are event pairs. Arcs for duplicate pairs are omitted. The figure shows a single 10-event window of events, but selection occurs for all 10-event windows.

To limit the number of constraints generated, we rely on the assumption that events that comprise effective constraints occur within a short window; we consider only event pairs separated by fewer than 10 events in the RPB. This assumption is reasonable for several reasons. First, prior work on finding and avoiding concurrency bugs [121, 85, 83, 128, 105, 66] suggests that the events involved in schedule-dependent failures often occur within a short window of the program's execution (*i.e.*, hundreds or thousand of instructions). Second, each event in the RPB represents a span of the program's execution, not a single instruction. Due to our online pruning approach (Section 7.3.2), two consecutive events in the same thread are at least 1μs apart, meaning that each event can represent thousands of instructions. Hence, a 10-event window covers a part of the execution large enough to contain useful event pairs.

7.4.2 Avoiding Failures

Each event pair corresponds to a schedule constraint. The first event in the pair is the constraint's "activation event", and the second event is the "delay event".

When a program instance starts, the framework makes a set of constraints available to the program instance. Every constraint starts as inactive. Inactive constraints have no effect on the program's execution. When a constraint's activation event is executed, the constraint is *instantiated*, and added to a set of active constraints. The runtime system records the ID of the thread that executed the activation event as the "activator" in the constraint instance. A thread may have at most one instance of a constraint active, but if several different threads execute a constraint's activation event, each thread will instantiate its own instance of the constraint. Figure 7.5 illustrates the constraint activation process.

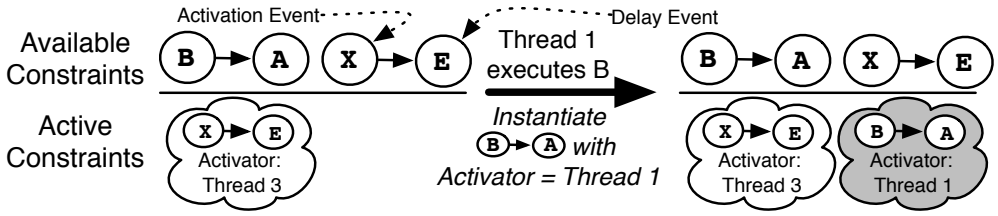


Figure 7.5: **Constraint Activation**. *Available constraints* are those that Aviso has made available to the execution. *Active constraints* are constraint instances that have been instantiated and can trigger delays. The large, central arrow signifies Thread 1 executing event B. To the left of the arrow there are no instances of the constraint (B, A); event B is its activation event, so when B is executed an instance of the constraint is added to the Active Constraints set (shaded cloud). Aviso records that Thread 1 is the instance's activator in the instance.

When a thread executes the constraint's delay event, Aviso decides whether to perturb the execution. To do so, it compares the executing thread to the activator of each constraint instance. If the thread executing the delay event is the same as the activator of a constraint instance, Aviso does nothing and execution continues. If it is different, Aviso delays that thread's execution. The delay perturbs the execution schedule by permitting threads other than the delayed one to continue their execution ahead of the delayed event. The reordering of these other threads' events with the delayed event is Aviso's strategy for preventing

failures, as we described in Section 7.2.

Practical Issues with Constraints

There are several practical issues related to pair-based constraints.

Delay Length. Events delayed by constraints cannot be delayed indefinitely without impeding forward progress. Delays must be long enough to reorder events that would lead to failures, but short enough that their impact on performance is tolerable. We empirically determined that $1ms$ achieved this balance well across our benchmarks: any shorter and Aviso was unable to prevent failures in some cases; any longer and performance degraded without improvement in failure avoidance. We show data in Section 7.7 that further support our choice of delay length.

Composing Constraints. It is important that Aviso not be limited to preventing only one failure due to one bug at a time. Most programs have more than one bug. Each bug may lead to a different failure. To deal with this problem, Aviso can make a collection of constraints available to threads, each with different activation and delay events. When any constraint's activation event is encountered, the executing thread instantiates that constraint. Threads can instantiate multiple different constraints simultaneously to avoid multiple different classes of failures. Section 7.5 describes how Aviso decides when multiple constraints should be available to be instantiated.

Why Do Event Pairs Make Effective Constraints?

Section 7.1 discussed the relationship between schedule-dependent failures and bug depth: if we invert one of the d event pair orderings necessary for a bug of depth d to cause a failure, we prevent the failure. In general, Aviso is effective if it generates constraints that reorder such events, like the one in the center part of Figure 7.6. We now describe how Aviso can use the constraint in Figure 7.6 to avoid two concrete classes of failures – atomicity violations and ordering violations.

Avoiding Atomicity Violations. Figure 7.6 shows how constraints avoid atomicity violations. It depicts two threads with Thread 1 executing events B and C, which should not

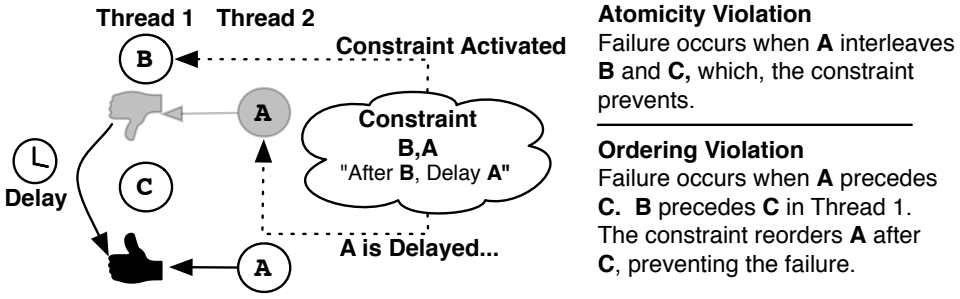
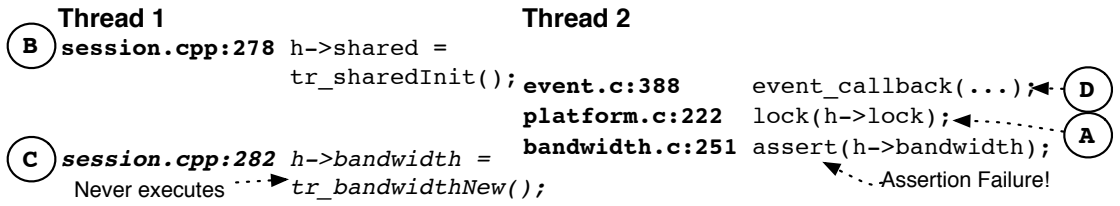


Figure 7.6: **How a constraint avoids a failure.** The constraint is shown in the cloud and is made from events B and A; when a thread executes B, the constraint is instantiated. When another thread executes A, it is delayed. The left side shows an execution snippet that can be viewed as both an atomicity violation and an ordering violation.

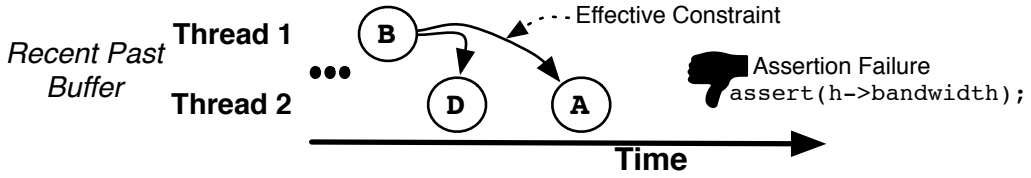
be interleaved by other events. Thread 2 is executing event A. The atomicity violation occurs if A happens between B and C. Note that there are two points in the execution where the failure can occur – just after A executes or just after C executes. In either case, the constraint prevents the failure. When Thread 1 executes B, it instantiates the constraint. When Thread 2 tries to execute A, it is delayed. During the delay, Thread 1 safely executes C. The delay prevents the failure by reordering A after C, rather than between B and C.

Avoiding Ordering Violations. Figure 7.6 also shows how constraints avoid ordering violations. To view the figure as an ordering violation, assume a failure occurs when Thread 2 executes A before event C.

Avoiding ordering failures is challenging because when the failure manifests, execution may fail just after C or just after A. If the program fails just after A, C will never execute and will therefore not appear in the RPB after the failure, so the Aviso framework will be unable to use it to form a constraint. To handle these failures, Aviso relies on the presence of a third event, B, executed by the same thread as C (Thread 1). In failing runs, B executes just before C would have and is added to the RPB. When A executes and the failure occurs, B is in the RPB followed by C. If C had executed, it would have immediately followed A. Hence, A following B in the RPB of a failing run indicates that the incorrect ordering of A and C is likely to have occurred.



(a) **A failing execution.** Events are identified by the labeled circles.



(b) **The RPB just after the failure.** Arcs indicate the event pairs Aviso enumerates and uses to generate constraints. The dashed arrow indicates that the pair (B, A) corresponds to a constraint that effectively avoids the failure by delaying Thread 2's execution of A until after Thread 1 executes B.

Figure 7.7: **A use-before-initialization failure from Transmission and the constraint that avoids it.**

The constraint in the figure is formed from B and A. When Thread 1 executes B, it activates the constraint. Later, when Thread 2 tries to execute A, it is delayed by the constraint. The delay gives C a chance to execute, preceding A. When the delay expires, A executes after C, avoiding the failure.

7.4.3 Constraint Generation Example: Transmission

Figure 7.7 illustrates a failure that can occur when Transmission-1.42, a multi-threaded bittorrent client, is starting up. Figure 7.7(a) shows an execution of the events that lead to a failure. The execution fails when Thread 2 reaches `assert(h->bandwidth)` at `bandwidth.c:251` before Thread 1 assigns `h->bandwidth` at `session.cpp:282`. In this situation, `h->bandwidth` is uninitialized, so the assertion fails.

Figure 7.7(b) shows an RPB snippet immediately after the execution fails, illustrating how Aviso finds an effective constraint. The situation in Figure 7.7 corresponds to the ordering violation situation in Figure 7.6. The effective constraint depicted delays A (Thread

2's lock acquire at `platform.c:222`), making it execute after C (Thread 1's assignment at `session.cpp:282`). This reordering prevents a failure because `h->bandwidth` is initialized by Thread 1 before the assertion executes.

7.5 Selecting and Distributing Constraints

After an execution fails, the Aviso framework generates a large set of candidate constraints to assess which can prevent failures. It *selects* the constraints most likely to avoid failures and *distributes* them to new program instances when they start up.

7.5.1 Selecting Constraints

Aviso selects constraints using a two-part statistical model. The first part of the model is the *event pair model* that represents properties of event pairs, as they occur in correct and failing executions. The second part, the *failure feedback model*, empirically determines which constraints are most effective by tracking each constraint's impact on the program's failure rate. As Aviso progressively observes more failing and non-failing program runs, its models improve, yielding better selections.

Event Pair Model

The event pair model represents each constraint with a vector of features. The value of each of these features is different for each constraint and is derived from execution behavior observed by Aviso in failing and non-failing executions. The magnitude of a constraint's feature vector determines how likely the constraint is to be effective.

There are two main concerns related to the event pair model: (1) Collecting the information that goes into building the model; and (2) Describing and computing the features' values for each pair.

Collecting Model Information The event pair model synthesizes information in RPBs from both non-failing and failing program executions. When Aviso collects an RPB, it updates the event pair model by recomputing each constraint's feature values.

The model uses the information in RPBs from failing program executions. In Section 7.4

we described how Aviso collects post-failure RPBs to generate constraints. Aviso uses those RPBs to update its event pair model. Aviso also collects RPBs from non-failing program executions. To do so, Aviso samples the state of the RPB very rarely during correct executions. At a uniformly randomized interval between 0.1s and 20s, Aviso interrupts execution, captures the state of the RPB, and uses it to update the event pair model. Note that if Aviso samples an RPB, just before a failure occurs, the RPB may contain events related to the failure. To keep these events out of its set of correct RPBs, Aviso waits a fixed period of 5s before incorporating the correct run RPB into its model. If in the interim the execution fails, the sampled RPB is discarded.

Features. Aviso represents each constraint with a vector of three features: *ordering invariance*, *co-occurrence invariance*, and *failure correlation*. Feature values are between 0 and 1. We engineered our feature representation so that larger feature values indicate a constraint is more likely to prevent a failure.

Ordering Invariance (OI) helps identify constraints whose events occur in one order in non-failing runs, but the opposite order in failing runs. Given a constraint built from a pair of events, (B, A), its $OI_{B,A}$ value is:

$$OI_{B,A} = \frac{\sum_{c \in CorrectRuns} f^c_{A,B}}{\sum_{c \in CorrectRuns} f^c_{A,B} + f^c_{B,A}}$$

where $f^c_{x,y}$ represents the number of times the pair x, y appears in the RPB sampled from correct run c. Note that the value of OI is larger if (A, B) occurs much more often in correct runs than (B, A). A large OI value suggests (B, A) is anomalous in correct runs and therefore related to the failure. Hence, perturbing the execution around (B, A) is more likely to avoid the failure.

Co-Occurrence Invariance (CI) identifies constraints whose pairs of events tend not to occur together in non-failing runs. Given a constraint built from a pair of events, (B, A), its $CI_{B,A}$ value is:

$$CI_{B,A} = 1.0 - \frac{\sum_{c \in CorrectRuns} f^c_{B,A}}{\sum_{c \in CorrectRuns} [\sum_{y \neq A} f^c_{B,y} + \sum_{x \neq B} f^c_{x,A}]}$$

Note that the fraction part of CI is large if B and A occur together frequently in non-failing runs, or if B and A occur with other events infrequently in non-failing runs. Both of these conditions suggest the pair (B, A) is *not* an anomaly in non-failing runs. We invert the sense of the fractional term subtracting it from 1.0. As a result, the value of CI is larger if B and A more often occur in non-failing executions in pairs with different events, rather than with one another.

Failure Correlation (FC) identifies pairs of events that occur frequently in failing executions. Given a constraint built from a pair of events, (B, A), its $FC_{B,A}$ value is:

$$FC_{B,A} = \frac{F_{B,A}}{F}$$

where F is the total number of failing executions and $F_{B,A}$ is the total number of failing executions in which (B, A) occurred at least once. A large FC value suggests that B and A tend to occur consistently as a pair in failing executions and are therefore related to the failure; therefore perturbing the execution around (B, A) is likely to avoid the failure.

FC is unlike CI and OI in two ways. First, CI and OI are computed based on RPBs from non-failing executions; FC is computed using data from failing executions. Second, unlike CI and OI, FC is computed using $F_{B,A}$ and F – numbers of executions, rather than numbers of occurrences of pairs (*e.g.*, $f^c_{B,A}$). This is because the frequency of a pair unrelated to a failure in a failing execution may be different because the execution terminated early due to the failure. Such differences act as noise in our model. Instead, our method for computing FC factors out this source of noise.

Failure Feedback Model

The second mechanism Aviso uses to select constraints is the failure feedback model. This model records the failure rate, FR, observed for each constraint. The model also records the failure rate with no constraints available. If the program's failure rate is low (*i.e.*, many non-failing runs, few failing runs) when a particular constraint is active, it is likely that that constraint helps avoid failures more than others.

Every time the program terminates, the failure feedback model is updated. If the pro-

gram exited normally, Aviso increments the model's record of the number of non-failing runs for all constraints available to the program instance during that execution. If the program fails, Aviso updates the model's record of the number of failing runs.

Dealing with Long-Running Programs To keep the failure feedback model up to date, programs send Aviso a message indicating success or failure when they terminate. However, long-running programs like servers terminate infrequently. If a constraint is effectively preventing failures, the program may run indefinitely. In this case, Aviso might *never* update the failure feedback model to reflect the success of the constraint. To handle long-running programs, we add a facility to Aviso to record "logical time" ticks. To use logical time, we rely on the programmer to add markers to the code that represent progress in the application. Each call sends the framework a message, telling it to increment the non-failing execution count of each constraint the program is using; hence, a logical time tick in a long-running program is effectively a "non-failing run". We found these calls trivial to insert, even into large and unfamiliar programs. For example, in our experiments with Apache and Memcached, we incremented logical time after 1,000 and 10,000 requests were processed, respectively.

Combined Avoidance Likelihood Model

The framework selects constraints by querying its *combined avoidance likelihood model*, which incorporates both the failure feedback model and the event pair model. The combined avoidance likelihood model is a probability distribution with an entry for each constraint. The value of a constraint's entry is the likelihood that it is effective as predicted by the event pair model, scaled by an exponential function of the constraint's observed failure rate, as recorded in the failure feedback model. Concretely, the amount of probability mass contributed by a constraint, (B, A) is:

$$P_{B,A} = \underbrace{(CI_{B,A} \times OI_{B,A} \times FC_{B,A})}_{\text{Event Pair Model}} \times \underbrace{e^{r_1 \times (1.0 - FR_{B,A}) - r_2} + s}_{\text{Failure Feedback Model}}$$

where r_1, r_2 are parameters of the exponential function used in the model and s is a

smoothing factor that keeps the model defined and bounded by 0 and 1. We chose $r_1 = 8$, $r_2 = 0.7$, and $s = 0.001$. These choices cause the function to peak when $FR = 0$ and bottom out at 0.001 (s).

The intuition behind the combined model is the following. The event pair model is *predictive* – the model's features encode our inductive bias, and data (RPBs) refine the model's predictions. The failure feedback model uses *feedback* to scale predictions made by the event pair model. The scaling factor varies exponentially with the constraint's failure rate – constraints that fail often are exponentially less likely to be selected than ones that fail rarely or never. As failures and non-failing runs occur, Aviso refines its models. Over time, effective constraints' probabilities in the combined model grow and ineffective constraints' probabilities decrease.

Figure 7.8 illustrates the combined model. Failure correlation (FC) is computed based on RPBs from observed failures. Co-occurrence invariance (CI) and Order invariance (OI) are computed based on RPBs sampled from correct execution. The failure feedback model maintains failure rate values for each constraint, computed by monitoring constraint failures. Aviso uses the combined model to select constraints according to a probability function composed of the event pair and failure feedback models, as shown.

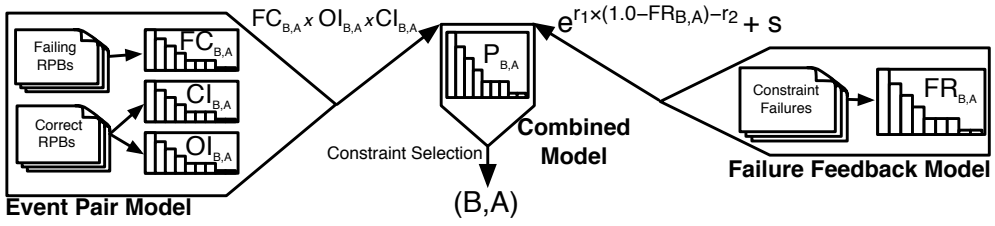


Figure 7.8: **Aviso's statistical model.** The event pair model tracks feature values for each constraint. The failure feedback model tracks constraints' failure rates. The combined model is comprised of the other two, yielding a selection probability for each constraint.

7.5.2 Distributing Constraints

All constraints start equally likely to be drawn. As program instances run, Aviso samples non-failing RPBs and stores them as the program runs. When the program fails, the framework generates constraints. The framework initializes the event pair model using the stored RPBs and the combined model assigns each constraint an initial probability based on the event pair model. The failure feedback model is ignored at this point because before any executions, all constraints' failure rates are undefined. Later, when a program instance starts, it queries Aviso for constraints. Aviso selects a constraint and sends it to the instance.

Aviso periodically instructs program instances to run with no constraints to establish the application's baseline failure rate. Initially, Aviso sends no constraint 10% of the time; the rate drops to 1% after seeing enough executions without any constraints to establish 95% statistical confidence that the observed baseline failure rate is within 5% of its actual value, assuming a binomial distribution of failures.

We continuously compute χ^2 statistics for each constraint to determine the significance of the difference of the constraint's observed failure rate and the baseline failure rate. We use a 2x2 contingency table and consider a difference to be significant if the χ^2 test indicates it to be with at least 95% probability. When a constraint that significantly decreases the failure rate is identified, Aviso uses that constraint 75% of the time. However, Aviso continues to draw constraints from the combined model for two reasons: (1) it is important to keep the baseline failure rate up to date; constraints may *lose* their significance if the baseline rate changes. (2) there may be other constraints with larger significant decreases in failure rate; halting its exploration, Aviso may settle for a non-optimal constraint.

Handling Multiple Failures

Up to this point, our discussion has assumed that all failures can benefit from a common pool of constraints. However, programs are likely to have more than one type of failure, stemming from more than one bug. Aviso also deals with multiple failures. The key is to maintain a separate model for each *failure class*. A failure class is identified by the content of the RPB at the point of failure. When a failure occurs, the post-failure RPB is compared

to the RPBs collected from failures in each failure class, using symmetric set difference. The RPB is assigned to the class to which it is most similar. If the set of events in the RPB is not at least 80% similar to the set of events in any existing failure class, a new failure class is created.

When a program instance starts and queries the framework for constraints, Aviso selects a constraint from each failure class according to its own model. The program instance is then sent one constraint per class. The constraint-less failure rate information is shared across classes. On a failure or successful run, all classes' models are updated.

There may be two unrelated failures that occur with similar RPBs. If the failures are assigned to the same class, only one constraint will be applied to starting program instances. As a result, it is possible that only one failure or the other will be prevented. We accommodate this situation by allowing Aviso to split a failure class in two if no constraint significantly decreases the failure rate after a fixed number of experiments. The purpose of this process is to select two constraints for what was previously a single failure class and thereby prevent both failures.

7.6 System Implementation

We built a complete implementation of Aviso including the profiler, instrumenting compiler, runtime system, and the constraint selection and avoidance-sharing framework. [1] We implemented the sharing profiler using Pin [87]. Dominance pruning and event placement were implemented in LLVM [73]. The framework and runtime were implemented from scratch.

7.6.1 Framework Implementation

The framework was implemented in about 3000 lines of Go code. The statistical models and constraint generation were implemented from scratch in the framework.

The framework exposes a messaging API. The API provides calls for the runtime to query for constraints, to send RPBs for sampled, non-failing periods and after failures occur, and

[1] Aviso is available for download at the authors' website.

to send logical time ticks. The API works over the network, via HTTP. The framework and runtime-enabled program instances form a distributed system that implements Aviso. Using HTTP as the messaging protocol for the distributed system makes it flexible, portable, and suitable for use in cloud environments such as Google AppEngine or Amazon EC2. Furthermore, its simple interface lets the framework be trivially replicated and lets replicas be load balanced for further scalability. Replicas' statistical models could be kept consistent via consensus or simply operate independently.

7.6.2 Runtime Implementation

We implemented the runtime in a library with an event handling API. Synchronization, signal, and sharing events make calls to the API. When a thread makes an event call, it records the event with a timestamp in a thread-local queue. When the thread ends, the timestamped events are serialized to a file. Timestamps are collected at nanosecond resolution using `clock_gettime`'s `CLOCK_MONOTONIC` clock. We use thread-local event queues and timestamps to collect events because they are faster than an earlier version of our system, which used a serializing event queue shared across threads.

Constraints are shared object plugins to the runtime. Each exposes a factory method to instantiate its constraint. When the program starts up, the runtime receives constraint descriptions as text. The runtime uses a simple, custom, templated code generator to produce C++ code from the text. The framework then calls out to gcc to compile the code to a shared library that is loaded by the program. Program instances cache compiled constraints, so code generation and compilation need be performed only once; subsequent executions that call for the same constraint can reuse the previously generated constraint plugin.

The runtime was built for concurrent performance. Its only shared data structure is the state associated with active constraints; everything else is thread-local. Accesses to the shared structure are rare: each thread has a thread-local list of events that may require an access to the shared structures. The common case is for a thread executing an event to check its list and continue without accessing the shared structure. Only when a thread

hits an event involved in a constraint must it consult the shared structure. This arrangement minimizes the chance that Aviso's data-structures will lead to serialization and poor performance.

7.7 Evaluation

We evaluate Aviso along several dimensions. First, we show Aviso's efficacy in avoiding failures. Second, we show that Aviso's overheads are reasonable both during data collection and when actively avoiding failures. Third, we characterize Aviso's constraint selection process. Finally, we characterize Aviso's dynamic behavior.

7.7.1 Experimental Setup

We evaluated Aviso using several buggy parallel programs.

Out-of-the-Box Benchmarks. Our main results are based on experiments with one cloud application, one server application, and one desktop application made to run with Aviso "out of the box".

Memcached-1.4.4 is an in-memory key value store with an atomicity violation that leads to data-structure corruption and lost updates under heavy update load. We added a single assertion that detects the data-structure corruption when a thread writes to a deallocated table cell and aborts execution. The data-structure invariant that our assertion checks is the cause of the lost updates, but the assertion is oblivious to the lost update problem; to write the assertion, a programmer would not need to understand the lost update failure. We manually added a single Aviso call to the server to send a logical time update every 10,000 requests. Inserting this call was trivial even without being familiar with the codebase. For profiling, we initialized a key-value store with 10 keys storing integers that 8 threads accessed. We used a mix of accesses that was 90% reads and 10% updates. For tests, we used a 10-key store and the same thread count and operation mix.

Apache-2.0.46 is a web server with atomicity violations in its in-memory cache subsystem that lead to crashes when concurrently servicing many php script requests. Our server setup is Apache with mod_php loaded, in-memory caching enabled, and serving the time

of day via a php script. We added a single Aviso call to send a logical time update every 10,000 requests. As with Memcached, inserting the call was trivial. For profiling, we used `ApacheBench` to issue 1,000 requests from 8 concurrent request threads for a static `html` page, then 1,000 requests from 8 threads for our php time server. For tests, we let the server run continuously until a failure. We sent time-of-day requests in groups of 10,000. To vary the workload, each group of requests was sent by a number of threads uniformly randomly chosen to be between 2 and 8.

AGet-0.4 (Figure 7.1) is a download accelerator with an atomicity violation in its signal handler that leads to output corruption. To detect failures, we manually added an assertion that aborts when it detects output corruption. The assertion compares a count of bytes written to the downloaded file to the sum of per-thread byte counts. The symptom of the failure is that these counts are not equal. Note that adding this assertion did not require understanding how to *prevent* the failure. We needed to understand only that the number of written bytes reported by AGet should match the number of bytes in the output file. To profile AGet, we downloaded a 1MB file using 8 threads from a local network resource twice, once letting it complete and once interrupting it with `SIGINT`. To test AGet, we started downloading a 700MB Linux image and interrupted the download with a `SIGINT` after 1s.

Schedule-patched Benchmarks. To further demonstrate Aviso's applicability, we conducted experiments with two other desktop programs. Unlike our first three benchmarks, we altered these two programs to amplify their failure rate. We applied patches that use sleep statements to lead execution toward failing schedules, similar to prior work [139, 140, 60]. These schedule patches are *not* essential – Aviso could be applied without them; we used them to facilitate experiments. Despite the patches, these results show Aviso's effectiveness for several reasons. First, the program is unchanged except for a single call to `usleep`. Second, the increased failure rate does not affect constraint generation or selection, except to reduce the time required for both. Third, events involved in the failure are identical in the patched and non-patched versions.

Transmission-1.42 (Figure 7.7) is a bittorrent client with a use-before-initialize error that leads to an assertion failure. To profile Transmission, we downloaded a Linux iso torrent without the schedule-altering patch. We ran tests on Transmission by running with the schedule-altering patch applied and downloading a non-existent torrent, which triggers the failure, causing a crash.

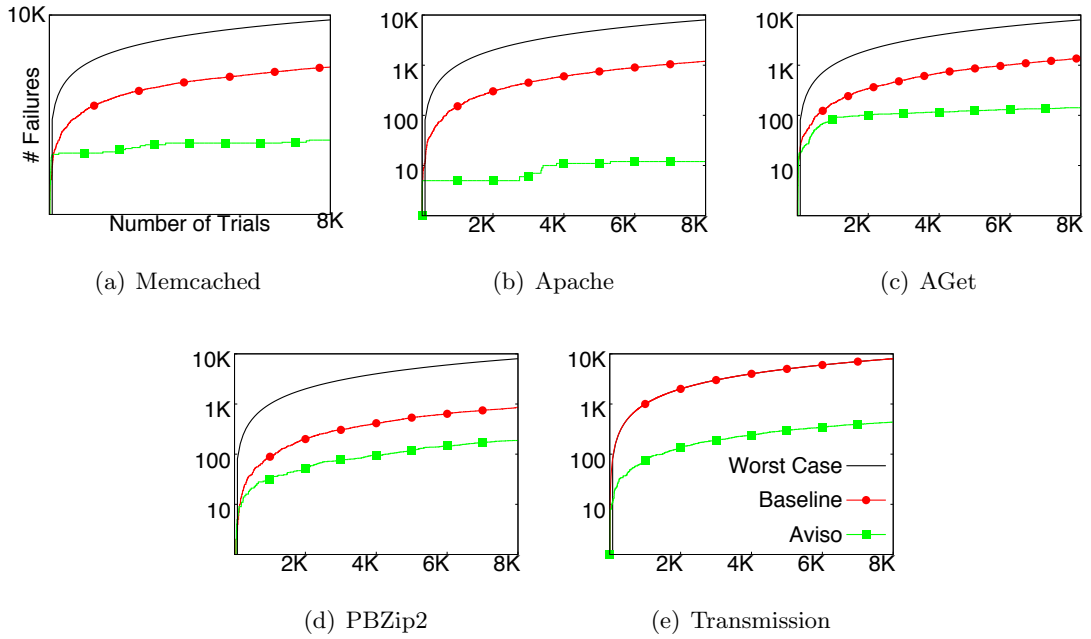
PBZip2-0.9.1 is a compression utility with a use-after-free error that leads to a crash. To profile PBZip2, we ran it under our profiler and first compressed, then decompressed, a 10MB text file. We experimented with PBZip2 by compressing a 250MB file. Aviso diagnosed the failure by watching for crashes and failed assertions.

7.7.2 Bug Avoidance Efficacy

Our main finding is that Aviso made our benchmarks fail less frequently, as shown on the plots in Figure 7.9. The plots show *on a log scale* the number of failures observed in our experiments for Aviso and for the baseline without Aviso, as well as a theoretical worst case. The slope at a point on a curve is the instantaneous failure rate at that point.

For all benchmarks, Aviso's curve is lower than the baseline, indicating a decrease in the number of failures experienced. In Apache's case, Aviso decreased the number of failures exhibited in our experiments by *two orders of magnitude*. Memcached saw a decrease in failures of more than an order of magnitude. Other cases had less pronounced decreases, but still benefited from Aviso.

These data elucidate how Aviso searches for the most effective constraint. For the first few runs of the program, the number of failures for Aviso is commensurate with the number for the baseline. During these first runs, Aviso is building and refining its statistical model. After a few runs, Aviso's model guides it to an effective constraint. At this point, the slope of the curve becomes flatter than the baseline, *i.e.*, Aviso begins to consistently choose a constraint or constraints that avoid failures.



(a) Memcached (b) Apache (c) AGet

(d) PBZip2 (e) Transmission

Figure 7.9: **Aviso's improvement in reliability.** We show data for (a)Memcached, (b)Apache, (c)AGet, (d)PBZip2, and (e)Transmission. The x-axis shows execution time in number of trials – logical time ticks for servers, executions for standalone applications. We ran each program for 8000 trials. The y-axis shows the the number of failures that have occurred at a given point in time *on a log scale*. The top (black) curve shows the worst case: every execution is a failure. The middle (red) curve shows the reliability of the baseline, compiled and run completely without Aviso. The bottom (green) curve shows the reliability with Aviso.

7.7.3 Performance

Table 7.1 shows that Aviso's runtime overhead is low. Column 2 is the overhead of event monitoring only. The overhead ranges from less than 3% for PBZip2 to 26.2% for AGet, with an average overhead of 13.4%. These results show that when Aviso is only collecting information, the performance overhead is tolerable.

Collection and Avoidance Overhead. Column 4 shows the combined overhead of event monitoring and avoidance. Avoiding failures does not prohibitively increase Aviso's overhead. For Memcached, the overhead is about the same as that of event collection alone; for Transmission, our worst case increase, the overhead is 20.7% greater than the overhead of event collection.

	Performance Overhead	
	Coll. Only	**Coll. & Avoid**
Transmission	8.8%	29.5%
AGet	26.2%	30.7%
PBZip2	2.6%	5.5%
Memcached	17.3%	16.7%
Apache	12.1%	15.9%

Table 7.1: **Aviso's runtime overheads.** These overheads are relative to baseline execution when collecting events only and when collecting events and avoiding failures.

PBZip2 has very low overhead for both event collection and avoidance because threads spend a majority of the execution in a compression routine in `libbz2`. Avoidance adds little to the overhead because the most effective constraint for PBZip2 involves events that execute during shutdown; constraint activation checks and delays need only occur during shutdown, so they do not impede the execution.

AGet's event collection overhead is high relative to our other benchmarks because a majority of the program's execution is in a tight loop that includes two event calls. AGet's avoidance overhead is only slightly higher than its collection overhead because the events involved in the constraints that Aviso found effective execute only during signal handling. The increased overhead is due to an increase in constraint activation checks, not delays.

Memcached's overhead is nearly the same for both collection and avoidance: the constraints that Aviso found effective are not activated in the program's common case. The events involved in effective constraints execute only when the number of digits in the number stored in one of Memcached's table cells increases, which occurs rarely.

The key finding, then, is that when collecting events only, Aviso imposes a low performance penalty. When avoiding failures, the overhead is only slightly higher.

Contrasting Improved Reliability with Aviso's Overhead The data show that Aviso's overhead is non-negligible. These overheads are acceptable for two main reasons. First, the increase in reliability comes immediately and without the need for the programmer to understand how to fix the program. Patches are hard to write correctly, and hand-written patches may introduce bugs or degrade performance. For example, Memcached developers

left the failure we studied unpatched for nearly a year after its initial report. They cited a 7% performance "regression" as one roadblock to committing a patch [36]. Aviso imposes a roughly similar performance overhead (16.7%) to the manually crafted solution and decreases the rate at which the failure occurs by nearly two orders of magnitude. Aviso does not require the programmer to understand how to fix the bug, let alone correctly patch the code to fix it. Furthermore, because Aviso operates automatically the gap between the first failure and Aviso's failure avoidance is a few minutes rather than the year required for the manual patch.

Second, Aviso's performance overhead saves programs from the potentially severe costs associated with failures. For example, Memcached's failure is a lost update that permits the store to be periodically corrupted but to continue executing. In safety-critical applications, data corruption is likely worse than performance degradation. Aviso provides the option of avoiding Memcached's data corruption at the cost of a modest performance hit.

7.7.4 Constraint Generation and Selection

Figure 7.10 characterizes how Aviso generates and selects constraints. Figure 7.10(a) shows that Aviso needs to experiment with only a small fraction of the constraints it generates to find effective constraints. Notice that the lower portion of the bars is considerably smaller than the upper portion: Aviso made only a small fraction of constraints available to program instances. Aviso effectively avoids failures, so this result shows that it selects effective constraints without having to observe many program instances.

Figure 7.10(b) shows that the number of constraints that Aviso makes available to program instances is small and for most of our benchmarks, most of the execution time was spent using a single effective constraint. These data reinforce the findings from Figure 7.10(a), *i.e.*, Aviso finds effective constraints after selecting and distributing only a few using its statistical model.

For Apache, Memcached, and AGet, the constraint represented by the bottom bar segment was used by Aviso for 92-99.7% of the execution time during our tests. For PBZip2, the bottom two bar segments account for nearly 80% of execution time; Aviso chose between

these two constraints a majority of the time. These frequently chosen bottom segments all represent constraints that led to a statistically significant decrease in failure rates.

In concert with Figure 7.10(a), this result illustrates how Aviso works: Aviso initially selects effective constraints without having to experiment with them or directly observe their impact on failure rates. It chooses good constraints without experimenting by using its predictive event pair model. After initially selecting a constraint that turns out to be effective (*i.e.*, the event pair model's prediction was a good one), the failure feedback model biases Aviso to select the same constraint again. The data directly show this phenomenon. For example, in Apache's case, Aviso selected 16 different constraints, experimented with each, and observed their impact on the failure rate. The 16th turned out to be effective, preventing nearly all future occurrences of the failure. Due to the constraint's success, the failure feedback model ensured it was subsequently selected.

Aviso selects and tests constraints differently for Transmission than for the other benchmarks. Transmission's lower 14 bar segments provided a significant decrease in the failure rate. Aviso used one of these 14 constraints for about 85% of execution time.

The data in Figure 7.10(c) explain why Transmission is different and help further characterize Aviso's event pair model. The bar height shows the ratio of the total number of constraints generated during our experiments to the total number of pairs in the event pair model that were observed in sampled correct RPBs. We call this ratio the *coverage* of the model. If the event pair model has fewer pairs than constraints – *i.e.*, has low coverage – it is likely to predict effective constraints poorly. If the model has more pairs, it is more likely to be useful in assigning meaningful selection probabilities to more constraints. Note that coverage may exceed 1.0 if pairs in the model never show up in a post-failure RPB.

Transmission's model coverage is zero. The structure of Transmission's failure explains why: the failure occurs very early in the program's execution. The event pair model is primarily built from RPBs sampled from portions of correct execution. Transmission crashes early, so no RPBs are sampled and the event pair model is of little use. Transmission's zero model coverage explains why Aviso was forced to experiment with more different constraints. Instead of predicting effective constraints, Aviso relied on the results of its experiments – the failure feedback model – to determine which constraints worked best.

Our other benchmarks had event pair models with higher coverage. Apache's model contained nearly the same number of pairs as there were constraints. Memcached and AGet also had models with high coverage. Looking back to Figures 7.10(a) and (b), the impact of higher model coverage is clear. For benchmarks with higher model coverage, the fraction of constraints used is lower and the fraction of execution time spent using a small number of effective constraints is higher.

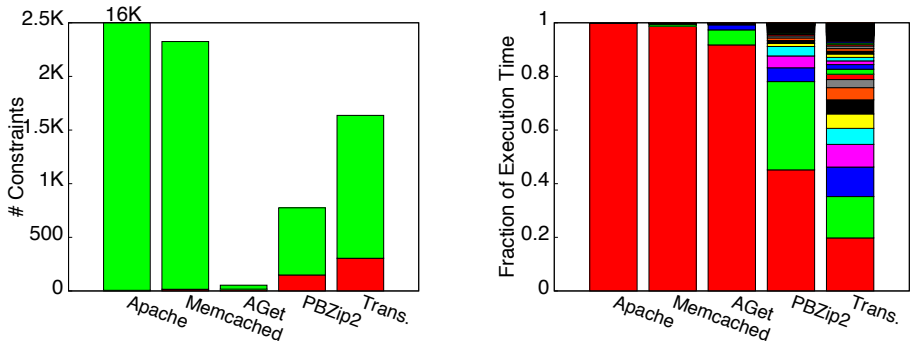
In summary, Figure 7.10(a–c) shows that when Aviso's statistical model has observed enough correct execution behavior, it makes good predictions, and Aviso is effective. If the model has low coverage, Aviso still selects effective constraints using its failure-feedback model.

7.7.5 Characterizing Dynamic Behavior

Using an instrumented version of the Aviso runtime, we characterized its dynamic behavior. For these experiments, we fixed a few runtime parameters: we chose the constraint used most frequently by Aviso during our main experiments, and we used a single fixed-size input. For PBZip2, we used the 250MB input file. For AGet, we downloaded a Linux image and interrupted execution (without crashing). For Apache, we issued 1M requests from 8 concurrent request threads. For Transmission, we downloaded a Linux image torrent, while for Memcached, we ran 80,000 client requests.

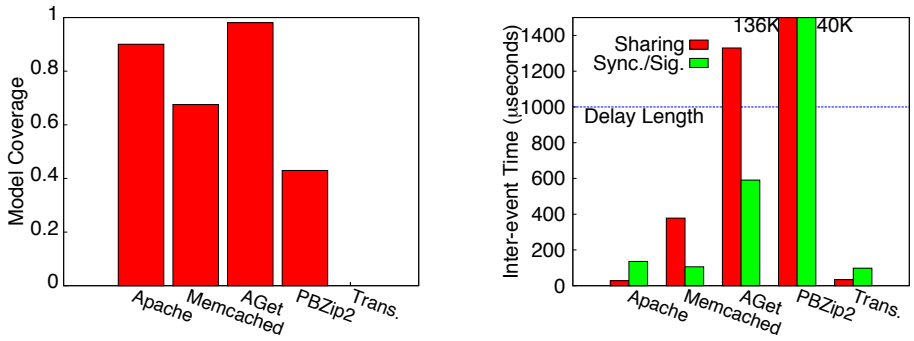
Figure 7.10(d) plots the rolling average time between events in μs during our experiments. These data justify Aviso's delay length. Recall that to avoid failures, events must be reordered by delays. In order to reorder events, events must be delayed long enough to allow an event in another thread to execute. The data show that the delay time is longer than the average inter-event time of most benchmarks. In the average case, a delay will reorder some events. Despite PBZip2's large average inter-event time, Aviso's delay length is adequate because the length of the interval between events involved in the failure is less than the delay length.

The data in Figure 7.10(d) also help explain Aviso's overheads. For PBZip2, the average time between events is several orders of magnitude larger than for other applications. The



(a) **Characterizing constraint generation.** The bar height represents the total number of constraints generated after any failure that occurred during our experiments. The lower, red segment shows the number of constraints that were actually made available to a program instance by Aviso.

(b) **Characterizing constraint selection.** Each bar corresponds to a different benchmark. Each segment corresponds to a different constraint. The height of a segment represents the fraction of execution time during vetting that ran with that segment's constraint. The total number of segments in a bar is the number of constraints Aviso experimented with.



(c) **Characterizing of Model Coverage.** Bar height represents model coverage: the ratio of the total number of constraints generated to the total number of pairs in the event pair model that were observed in sampled correct RPBs.

(d) **Characterizing inter-event time.** Bars show rolling average time between events in μs. The left (red) bar reflects sharing events and the right (green) bar, synchronization and signal events. Longer intervals mean events are less frequent. The horizontal line shows Aviso's delay length.

Figure 7.10: **Characterizing Aviso's behavior.**

length of the interval makes sense because PBZip2 spends most of its time in a compression library call. It is likely that PBZip2's long time between events contributes to the low overhead reported in Table 7.1. Each of the other benchmarks has a shorter interval between events than PBZip2. Correspondingly, the event collection overheads in Table 7.1 for the

	Sharing Events		Sync/Sig Events		Constraints		
	# Evts	# Discard	# Evts	# Discard	# Chks	# Strt	# Delays
Apache	44.6M	37.9M	1.5M	34K	27K	65	56
Memcached	200K	13K	1.2M	204K	87K	16	87
AGet	46K	0	92K	2	46K	5	40
PBZip2	227	0	1042	4	81	8	8
Transmission	10.6M	5.0M	2.0M	5	96	1	1

Table 7.2: **Aviso's dynamic behavior.** Columns 2 and 3 show the total number of sharing events and the number of sharing events discarded due to online pruning. Columns 4 and 5 show the total number and number discarded of synchronization and signaling events. Column 5 shows the number of times an event in an available constraint executes, requiring a check to see if the event activates the constraint. Column 6 shows the number of times a check actually leads to a constraint's instantiation. Column 7 shows the number of times an event is delayed by a constraint.

other benchmarks are slightly higher than PBZip2's.

AGet, with the highest performance overhead, has several events in its inner loop. We expected these events to result in a short interval between events, explaining its overhead. However, as Figure 7.10(d) shows, AGet's inter-event interval was moderately *longer* than most other cases. Table 7.2 shows data that explain AGet's performance and further characterizes Aviso.

The data in the table illustrate several different sources of overhead: excessively frequent events, leading to many discarded events; constraint activation checks; and delays due to constraints. AGet's overhead is likely to come from frequent constraint activation checks. In AGet, 1 out of every 3 events requires the extra computation of a constraint activation check. In contrast, 0.4% of Apache's and 7.25% of Memcached's events required such checks. Constraint activation checks require holding a lock and accessing shared state, so they are more costly than those that do not.

The data in Table 7.2 show that a large fraction of Apache's events (over 80%) were discarded due to online pruning. The high rate of discards suggests that events are frequent; the very short inter-event time shown in Figure 7.10(d) corroborates this fact. Intuitively, such high-frequency events seem like a performance problem; however, Apache's event frequency did not impose excessive overhead – around 15%. Most of Apache's events did not require activation checks, instantiations, or delays. As a result, its events were inexpensive, requiring just a few accesses to thread-local memory. The absence of complex computation

or serialization on global state is likely the reason for Apache's low overhead.

Delays were very infrequent across all our benchmarks, occurring mostly in uncommon case code. In PBZip2, 8 worker threads delayed a cleanup thread. In Transmission, a delay during startup code prevented a use-before-initialization error. In Apache, a delay during a request cache flush prevented a crash. In AGet, a delay during signal handling prevented a crash. In Memcached, a delay during a rare-case update prevented data corruption.

To summarize, delays were not a problem in our tests because they were infrequent and in rare-path code. Event frequency alone did not dictate performance, although having very infrequent events seemed to lead to lower overhead (*e.g.*, PBZip2). Constraint activation checks seemed to be a more costly source of overhead than we expected, especially when events were frequent (*e.g.*, AGet).

7.8 Conclusions, Insights, and Opportunities

We presented Aviso, a system that automatically avoids concurrency-related program failures by empirically determining fault-free execution schedules. Aviso leverages a community of instances of a program and uses statistical techniques to quickly determine which program events (and their order) are the culprit of a failure. We built Aviso in software only (not relying on special hardware) and our evaluation showed that Aviso increases reliability significantly (orders of magnitude reduction in failure rate in some cases) and leads to overheads acceptable in real production runs.

Insights. There are several major insights in this chapter. First, we showed that schedule perturbation can avoid schedule-dependent failures in a *general* way. Second, we showed that careful execution monitoring is essential to efficiently collecting data required to identify parts of an execution schedule that lead to failures. Third, we showed that there is "wisdom" even in crowds of machines: incorporating event histories from various failing executions yields enough information for Aviso to automatically generate and isolate effective schedule constraints. Fourth, we showed that in some cases (*e.g.*, Memcached) the overhead of automatic failure avoidance is similar to the overhead of a manual patch, but that the time to avoidance is far lower in the automatic case.

Opportunities. This chapter also presents several rich veins of future research. Aviso uses a rudimentary statistical model. A model in future work could incorporate quality of service criteria, such as throughput. Doing so enables balancing failure avoidance effectiveness with overall system performance. A model in future work could also rely on static analysis to better identify parts of code that are more or less likely to be related to a failure. Machine learning over information extracted by a static analysis of the program could be a fruitful approach.

Aviso's schedule constraints are very simple and could be improved in future work. The key limitation of Aviso's schedule constraints are that they use a delay to perturb the schedule. A delay-based mechanism eliminates the risk of causing deadlock, but also provides no guarantee that failures will be avoided. Future work could develop an analysis that finds not just pairs of events around which the schedule should be delayed, but rather precisely identifies when a fruitful perturbation has occurred. Determining when perturbations have occurred requires inter-thread coordination, which could be expensive, and more sophisticated code analysis to identify which events should be involved.

Aviso also presents several limitations in terms of security and privacy. Aviso's constraint selection model is subject to potential "poisoning" attacks by adversarial members of a community of software instances. An attacker may know that some schedule perturbation *exposes* a security exploit. That attacker may then send a large set of curated event histories to Aviso. The set of histories, if cleverly designed, could lead Aviso to generate constraints that are likely to expose the exploit. Privacy is also a concern that Aviso does not address. Systems using Aviso are required to send event histories back to the Aviso server. While event histories do not reveal information about data being processed by a program, code point histories may be adequate to reveal sensitive information.

Aviso's performance overheads, while low enough for many production environments, are high enough to be problematic in some situations where latency is critical. There are many ways Aviso's performance overheads can be reduced. One way is to more aggressively prune instrumentation, focusing on only instrumentation that is determined to be important for avoiding failures. Sampling instrumentation points may prove beneficial. Another especially exciting future research direction for reducing the overheads associated with Aviso is

to explore the role of hardware support. General hardware support for executing a pair of events in a particular order would be of use to Aviso for implementing constraints. Such a mechanism may also be useful for implementing reactive programming models, synchronization libraries, or other language-level features. Hardware support may also aid in reducing the overhead associated with monitoring events in Aviso. By tracking memory operations at the level of the hardware, the software overheads (*e.g.*, cache effects, additional code) of tracking are eliminated. In addition, a hardware mechanism for event tracking is also likely to be useful for a variety of applications, including bug detection techniques (*e.g.*, Recon) and other failure avoidance mechanisms (*e.g.*, Atom-Aid and ColorSafe).

Finally, as future work, Aviso may also be useful as a component of a *synchronization synthesis engine*. Schedule constraints are a simple, partial form of synchronization. By running a program written without synchronization under Aviso, schedule-dependent failures exposed during testing could lead to many Aviso-generated schedule constraints. These constraints, in aggregate, may be enough to make the program run correctly in a concurrent environment without the need for the programmer to spend time and effort writing synchronization code. Combining information about QoS, like throughput, might provide additional value, enabling synthesis to choose constraints that minimize performance overhead.

Chapter 8

RELATED WORK

Prior work related to this dissertation falls into two main categories: work on *debugging and bug detection* and work on *failure avoidance*. The discussion of work in each category further breaks work down according to its scope or mechanism. This chapter is not comprehensive. Instead, in each category, this chapter aims to include a selection of foundational papers and some state-of-the-art papers. The goal of including foundational work is to set the context for current work. The goal of including state-of-the-art work is to draw timely comparisons to the work in this dissertation.

For each prior effort that is covered, this chapter provides a brief summary to facilitate reading without needing to refer to prior work. This chapter then compares each prior technique with the work in this dissertation, highlighting similarities and differences. Table 8.1 illustrates the categorization of related work used in this chapter.

Overview of Related Work by Category

Debugging	**Avoidance**
Data-Race Detection	Data-Race Exceptions
Atomicity Violation Debugging	Avoiding SC Violations
Communication-Based Debugging	Synchronization Synthesis
Event-Sequence-Based Debugging	Avoiding Atomicity Violations
Multi-variable Errors	Online Patching and Patch Synthesis
Exposing and Reproducing Failures	Schedule Memoization
Statistical Debugging	Determinism

Table 8.1: **Categories of prior work discussed in this chapter.**

8.1 Debugging Concurrency Errors

The key challenges to debugging concurrency errors lie in understanding variations in inter-thread interactions and event orderings in a failing execution schedule. Prior work has looked at these problems in different ways.

8.1.1 Data-Race Detection

Early work developed fundamental abstractions for understanding ordering in concurrent computation. Lamport's *Happens-Before* relation provides a framework for analyzing the order of events in a distributed system [70]. Subsequent work by others developed the notion of *vector clocks* [43, 91], capturing a better partial order that includes orders implied by causality. Lamport also developed the definition for SC [71] in early work. Happens-before and SC are fundamental properties of concurrent software and are relied upon by a tremendous amount of subsequent work in the field. Lamport's work is directly relevant to error detection. The most common formulation of a data-race defines data-races using the happens-before relation.

Following Lamport's work, there have been many different approaches to data-race detection – some software and others hardware; some static and other dynamic; some precise and some approximate.

Precise Software Data-Race Detectors Early dynamic approaches to data-race detection were focused on restricted classes of programs, such as those written in Fortran without nested parallelism [58]. These techniques directly computed the happens-before relation over memory accesses to find races.

FastTrack [45] is the current state of the art precise (no false positives, no false negatives) happens-before data-race detector. Fast-track uses specialized vector clocks to track the happens-before relation for memory accesses. FastTrack gets high performance because it does not track a full vector clock for data that have not been read-shared since their last write. The authors prove that their technique detects at least the first data-race in an execution precisely.

Adve *et al* [13] discuss the extension of precise data-race detection techniques to systems with relaxed memory semantics. Their work formalizes the semantics of a precise, happens-before data-race detector in an execution with races, showing that it is possible to guarantee at least the first race will be correctly reported, under certain reasonable assumptions about the system.

Hardware Data-Race Detectors Min, *et al* [93] use hardware support to detect data-races in executions of fork-join parallel programs. The described system adds meta-data to each cache line that abstractly encodes the time of its last write. The meta-data, coupled with information gleaned from the exchange of cache coherence messages is sufficient to identify some racy memory accesses. With bounded size caches and imprecision introduced by cache line aliasing, the technique approximates happens-before, enabling the detection of racy accesses.

Around the same time as Min *et al*, Gharachorloo *et al* [51] developed a technique that tracks the completion order of in-flight memory accesses on relaxed memory model architectures. The goal is to conservatively determine when re-ordered memory accesses may have caused a violation of sequential consistency.

ReEnact [111] uses hardware support designed to support Thread-Level Speculation (TLS) to implement a low-overhead data-race detector. ReEnact works by using the ordering on TLS epochs induced by synchronization to determine if two memory accesses in the program are concurrent and constitute a data-race. The mechanism effectively implements a vector clock using the TLS versioning and conflict detection support.

CORD [110] detects data-races using hardware support that computes an approximation of the happens-before relation for accesses to shared memory. The mechanism relies on *scalar clocks* (as opposed to full vector clocks), a small set of timestamps added to cache lines and a main memory timestamping mechanism. CORD is similar in spirit to ReEnact, though it trades off some precision to reduce the complexity of the design.

Lockset Data-Race Detection Eraser [119] is a *lockset* based race detector. The key idea behind lockset-based techniques like Eraser is that the set of locks held when a shared

memory location is accessed should be consistent from one access to the next. Eraser tracks the set of locks held by a thread when a memory location is accessed. If there is no lock that is consistently held across all accesses to the memory location, Eraser reports that there is not a consistent locking discipline governing accesses to the location. Lockset analysis does not require a race to occur in order to report it to a programmer – only that the locking discipline is inadequate to ensure races do not occur. Reporting *potential* races when locking discipline violations are detected is a strength of lockset detectors. The downside of lockset analysis is that it reports some false positives. Lockset analysis has been combined with happens-before analysis in hybrid software race detection analysis [141, 108]. HARD [145] is a hardware implementation of lockset race detection analysis.

Comparison with this work Data-races are a source of execution schedule variation and are often concurrency bugs. Data-races are the first type of concurrency bugs that were widely studied, serving as a jumping-off point for other research on concurrency bugs, including the work in this dissertation.

Data-race detectors help programmers understand non-local effects in a program by revealing pairs of unordered operations. Racing pairs of accesses can interact in unintuitive ways, subject to the memory model of the system executing the program. As Bugaboo and Recon makes non-local interactions between communicating accesses explicit, race detection makes non-local memory ordering interactions between racy accesses explicit.

Several of these mechanisms are similar to the work in this dissertation in their implementation requirements. As Bugaboo uses per cache line meta-data to track last writers, Min *et al* [93] tracks access sets using cache line extensions. Their mechanism is relevant because Min *et al* predates our work by twenty years and solves a related problem using a similar mechanism.

Like Bugaboo's hardware implementation, Atom-Aid, and ColorSafe, Min *et al* also used the coherence protocol to monitor thread interactions. Cache coherence tricks have been used in many systems, notably those supporting transactional memory, but the use in this prior work is especially relevant because it is for concurrency bug detection.

ReEnact and CORD are mentioned because they track happens-before memory ordering in hardware during an execution. ReEnact uses complex multi-versioning TLS support.

CORD tracks happens-before without TLS support, but is approximate and limits its race detection to cached data – precision is lost when data are evicted from the cache. These systems' implementations of dynamic concurrency analysis in hardware is like the ones used in collecting context-aware communication graphs.

Lockset-based techniques are relevant to concurrency bug detection in general. Furthermore, some lockset techniques have the interesting property that using *dynamic* analysis, they can detect data-races that could, but did not, occur in a monitored execution. Our goal in Atom-Aid and ColorSafe is detecting and avoiding *potential* errors and our pursuit of that goal was inspired in part by lockset race detectors.

8.1.2 Atomicity Violation Debugging

Artho *et al* [14] was among the first efforts to characterize atomicity violations for shared-memory programs, calling them *high-level data-races*. In that work, a high-level data-race is the result of a programming error that fails to ensure the atomicity of memory accesses that should have been atomic. The authors introduce criteria for finding such errors by inferring atomicity constraints intended by the programmer. The analysis identifies *views*, which are sets of memory locations accessed in the same locked critical region. A thread's views are computed and compared to views collected by other threads. If one thread accesses data together in the same view and another thread accesses the same data separately in different views, a *view consistency violation* is reported. The presence of a view consistency violation implies the presence of an atomicity violation.

Researchers have investigated static analysis to detect potential atomicity violations and prove the absence of atomicity violations [47, 118, 130]. The techniques rely on a type and effect system for expressing and checking the atomicity of methods in a program. The authors prove that if a program typechecks under their type and effect system, there can be no violations of the methods' specified atomicity properties. The authors use Lipton's theory of reduction [76] and static race detection together to check the atomicity properties.

Several dynamic approaches to checking for atomicity violations also exist. Flanagan *et al* [44, 48] develop dynamic analyses to check atomicity properties. Atomizer [44] uses

modified lockset analysis to detect when the atomicity of atomic blocks is violated during an execution. Velodrome [48] analyzes execution traces and computes the *extended happens-before* relation for operations. Extended happens-before combines conventional happens-before with the execution order equivalence of operations in the same transaction. Both dynamic analyses report atomicity violations, Velodrome with higher precision and Atomizer with higher recall [48].

SVD [136] is another dynamic approach to detecting atomicity violations. Unlike the dynamic analyses discussed already, SVD does not require an atomicity specification (*i.e.*, `atomic` annotations). Instead, SVD infers *Computational Units* (CUs) that approximate atomic regions. CU inference is heuristic, considers control and data-dependences, and does not rely on specified synchronization. SVD then uses serializability analysis based on two-phase locking to detect the presence of atomicity violations with respect to the inferred CUs. These potential violations are then reported to the programmer.

AVIO [80] is a technique that detects pairs of instructions that execute atomically, without another thread unserializably interleaving an operation between them, during a set of training runs. The set of pairs of instructions are called *access interleaving invariants* (AI Invariants). AVIO builds up a set of AI Invariants from correct program executions and then monitors during subsequent executions for violations of those AI Invariants. Unserializable interleavings of AI-Invariant instruction pairs indicate a likely atomicity violation bug, which is reported to the programmer. The authors show that AVIO can be implemented in hardware with high performance or in software with lower performance than the hardware implementation, but higher precision.

AtomTracker [97] is an atomic region inference technique and atomicity violation detector. Like AVIO, AtomTracker monitors a set of executions and infers atomic regions based on observations of correct executions. Unlike AVIO, AtomTracker considers atomic units larger than instruction pairs. It uses a trace-based instruction merging analysis to determine atomic groups of instructions. AtomTracker then uses the inferred atomic regions in a dynamic analysis that identifies atomicity violations.

AtomFuzzer [103] uses a very simple heuristic technique for identifying atomicity violations. AtomFuzzer assumes that if the same lock is acquired, released, and re-acquired in

the same function, an atomicity violation is likely to have occurred. The heuristic hinges on two assumptions: (1) between a release and an acquire in the same function other threads are likely to interleave accesses; and (2) accesses split across two critical regions in the same function probably should have been written to execute in the same critical region.

Comparison with this work

Atomicity violations are an important class of errors, singled out by Atom-Aid and ColorSafe and dealt with by Aviso, Bugaboo, and Recon. Several prior techniques [97, 80, 136, 48] use serializability analysis, which we make heavy use of in this work.

These prior techniques illustrate a wide variety of methods for inferring atomicity of operations in a program, using program structure, dependences, serializability, and error-specific heuristics. Many differ from our work in the way they infer which code should be atomic. AVIO finds pairs of accesses to the same memory location that are never interleaved in training runs and assumes they should be atomic. AtomTracker extends AVIO's analysis to finds sequences of accesses that were never interleaved during training. SVD and AtomFuzzer take a *code-centric* approach, using dependence information and heuristics to determine atomic blocks.

The inference heuristic we use in Atom-Aid and ColorSafe differs from prior work. Our technique infers atomicity of fixed-length sequences of instructions beginning with an access to data previously involved in an unserializable interleaving. Our technique was inspired by the pair-based approach used in AVIO that was used later adapted by AtomTracker. We generalize AVIO's heuristic from pairs to arbitrary, fixed-length sequences and specialize the heuristic to focus on sequences beginning with an access to certain suspect data. Heuristics like ours and AVIO's are simple and program semantics oblivious. The simplicity leads to some false positives, but affords straightforward implementation in hardware. False positives are bad because they waste programmer time, but we show that using post-processing, we can eliminate many false positives. AVIO takes a similar step, relying on invariance over many executions to deal with false positives. Additionally, our main focus in Atom-Aid and ColorSafe is avoiding failures, not debugging. False positives may decrease run time performance by triggering spurious avoidance actions, but will not waste programmer time.

Secondly, our debugging approach in Bugaboo and Recon is more general than these

approaches. Our work *includes* support to detect atomicity violations, but also addresses other types of errors. The key to our approach is that instead of tracking atomicity directly, we track communication. Failures due to many types of errors including atomicity bugs manifest as communication graph anomalies. Our generality is a key advantage over bug-specific techniques.

Thirdly, AVIO and AtomTracker build a rigid model of program behavior – if an invariant pair or atomic unit is observed during training executions, it is inferred to be an access interleaving invariant. These models do not incorporate behavior from failing runs. Information from failing runs is potentially valuable in determining which operations should be atomic. Furthermore, the models do not track the *frequency* with which each inferred atomicity constraint is expressed in program execution. A model that tracks frequencies can associate a confidence measure with each inferred atomic region. In contrast to AVIO, Bugaboo and Recon build a more flexible model of program behavior. Our model incorporates information from both failing and non-failing program executions. Our model uses continuous-valued features derived from observed executions, rather than binary invariants, as in AVIO.

Aviso's model takes an even more general and flexible view of atomicity. Aviso deals uniformly with atomicity violations and other bugs, using an analysis rooted in the concept of bug depth [25]. Aviso is more general because it includes atomicity violations, but is not limited to them. Aviso is more flexible, because it incorporates information from many executions, failing and non-failing, and its statistical model adaptively determines what code is likely responsible for a failure.

8.1.3 Communication-Based Debugging

Prior approaches that use information about inter-thread communication to identify bugs are closely related to Bugaboo and Recon.

DefUse [121] analyzes invariants over the flow of data from definitions in one thread to uses in another thread to detect bugs. The authors reason from a set of three patterns of data-flow between threads, each tailored to handle a particular failure mode: the first

for some types of atomicity bugs and ordering bugs, the second for the remaining types of atomicity bugs, and the third primarily for sequential bugs. The authors train a model of program behavior consisting of these invariants. The technique detects violations of the invariants and reports them, ranked by an ad hoc ranking function.

DMTracker [50] is a technique for finding inter-thread communication bugs designed for MPI programs. The main idea of DMTracker is to monitor many program executions and build up a set of *data movement* (DM) invariants. DM invariants encode how data flows from one thread to another, between a pair of instructions. The technique uses a training phase to identify DM invariants and then flags violations of the invariants as potential errors.

Comparison with this work

These techniques are included in this chapter because they are related to Bugaboo and Recon in their goal and their approach. There are several fundamental differences, however.

While DefUse uses communication information, it is essentially *pattern-based*, relying on a small set of patterns tailored to only certain kinds of ordering and atomicity errors. Bugaboo and Recon are not pattern-based, instead relying on the invariance of arbitrary parts of a communication graph. The lack of reliance on *ad hoc* patterns permits our technique to detect a more general class of errors than can be detected using DefUse's invariants. One class of errors our work can detect that DefUse cannot detect are errors involving multiple pairs of communicating accesses to multiple different memory locations. A pattern could be added to DefUse to handle such cases, but the approach does not generalize well.

DMTracker, while not pattern based, is fundamentally limited in the class of errors it can detect. The main limitation is that DM invariants encode only communication. We show in Chapter 2 that many errors cannot be detected using communication alone. Our solution was to add communication context to communication graph nodes, distinguishing different dynamic instances of instructions. Our work was inspired by DMTracker, because our basic approach is similar to the DM invariant concept.

Another contrast between DefUse, DMTracker, and our work is in what is reported to the developer and why. DefUse and DMTracker report the inferred invariant that was violated to

programmers because violated invariants are likely to be related to the failure. In Bugaboo, our goal was similar – Bugaboo aims to report communication anomalies that are related to a failure. Recon, however, furnishes the programmer with a *reconstruction* of a focused portion of the program's execution around a communication event inferred to be relevant to an error. Recon essentially solves a different problem than DefUse, DMTracker, or Bugaboo: rather than identifying a pair of communicating code points related to a failure, Recon builds a model of program behavior and generates a likely execution fragment surrounding pairs of communicating accesses. Recon then uses the model to infer which *reconstructions* are most likely related to the failure.

DefUse and DMTracker are based on software instrumentation and data collection. Due to its complexity, it is unlikely that DefUse could be made efficient enough for always-on deployed use. The authors of DMTracker report that it is efficient enough for always-on use with only a small run time overhead on a set of scalable HPC benchmarks. It seems unlikely that the technique can be made as efficient on less scalable benchmarks, or programs that use less structured concurrency constructs (*i.e.*, not MPI). In contrast, our initial work that developed the context-aware communication graph abstraction proposed both a software implementation with debugging-time use in mind and a hardware implementation with always-on use in mind. We show that with hardware support, collecting communication graphs can be made efficient enough for use in deployed software.

8.1.4 Event-Sequence-Based Debugging

ConSeq [143] is a concurrency error debugging technique similar to Bugaboo and Recon. ConSeq works by using static analysis to identify potential failure sites in a program, (*e.g.* assertions, etc). ConSeq then statically looks at short backward program slices starting from each failure site. The goal is to capture *critical reads* in the slice. Critical reads read data that propagates to the failure point, leading to the failure. ConSeq collects multi-threaded traces and tries to perturb the multi-threaded execution schedule to find new, feasible traces that would change the value obtained by the critical read and potentially cause a failure. If such a perturbation, critical read, and failure point are located, they are reported to the

programmer as a potential error.

ConMem [144] is a concurrency error debugging technique very similar to ConSeq [143]. ConMem focuses on concurrency errors that lead to memory errors (null pointer dereferences, etc). The key idea to ConMem is to examine multi-threaded traces and, using a set of perturbation patterns, identify perturbations of the trace that could lead to a memory-bug-related failure. The tool then reports the pattern and the operations involved to help a programmer understand the error.

Falcon [104] works by examining an execution and isolating interleaving patterns from a set of patterns corresponding to common concurrency-related failures. The tool records how frequently instances of a pattern occur in failing and non-failing runs of the program. Using this frequency information, Falcon isolates interleaving sequences that occur often in failing runs and not often in non-failing runs. These sequences are reported to the programmer.

Comparison with this work

ConSeq is most related to Recon. There are several key differences. First, our work and ConSeq solve different problems. ConSeq develops machinery to perturb an execution trace to produce a new trace that represents a failure. The potentially failing trace is exercised to try to produce a failure and information about the failure is reported. In contrast, Recon focuses on debugging an observed failure. Recon starts from a failed test case or bug report. Recon composes snippets from failing executions likely to be related to the failure and reports them to help the programmer see what went wrong. The difference between ConSeq and Recon is that ConSeq aims to find new failures and Recon focuses on helping programmers understand reported failures. Recon statistically models a set of failing executions. ConSeq, instead, works from a single execution and generates feasible failing executions.

Second, ConSeq must limit its static analysis component, due to the large space of slices and critical reads. In contrast, when Recon builds reconstructions, it only needs to consider the space of events observed in failing executions. Recon bounds a reconstruction's size, but only to limit the amount of noise in the output reconstruction, not because of the performance of our algorithm. Recon can therefore consider including events in a reconstruction that occurred in a span of instructions that ConSeq cannot capture in a bounded slice.

ConSeq and ConMem are related to Aviso because all of these analyses perturb observed event sequences to assess their impact. Aviso analyzes perturbations to find alternate sequences that prevent failures. ConSeq and ConMem analyze perturbations to find feasible execution schedules that can lead to failures.

ConMem and Falcon are related to Bugaboo and Recon as well, but with several essential differences. ConMem and Falcon are both pattern-based, relying on a library of patterns associated with types of memory-related failures (ConMem) and access interleavings (Falcon). Pattern-based reasoning, while simpler to characterize and implement, is fundamentally less flexible than communication-based analysis: any failure not represented by an included pattern will not be detected. Note that while communication-based analysis will miss any failure that does not manifest in a CACG or reconstructions, we show that the class of errors we detect subsumes the class dealt with by ConMem and Falcon.

Falcon is also similar to Bugaboo and Recon in its statistical modeling of pattern instances. Falcon's model includes information collected from both failing and non-failing executions to quantify the "suspiciousness" of each pattern instance. Bugaboo and Recon also build a model using information from failing and non-failing executions. Unlike Falcon's single-featured model, our model uses several features of the executions we analyze, incorporating information about communication context, common instructions sequences, and the frequency of communication events.

8.1.5 Multi-variable Errors

Prior work has detected programming errors involving accesses to multiple different memory locations. These papers are included in this chapter because in ColorSafe, we focus on the multi-variable atomicity violation problem. Furthermore, in Bugaboo and Recon, we aim to address single- and multi-variable bugs.

Early work on atomicity violations did not distinguish between single- and multi-variable concurrency bugs, referring to all of them simply as *high-level data-races* [14]. This prior work addresses high-level data-races by computing "views", which are multi-variable groups that should be updated atomically. These groups are the same as the ones used in ColorSafe

to avoid multi-variable atomicity violations.

Researchers have used *atomic-set serializability* [55] to identify potential concurrency errors. Atomic-set serializability is a generalization of serializability from individual memory locations to sets of locations. In their initial work on the subject [127], the authors identified 11 patterns of accesses to multiple related memory locations that can lead to unserializable behavior. The set is *complete* in the sense that all method executions will be serializable given that none of the 11 patterns appears in the execution. In later work [55], the authors develop a very similar dynamic analysis that detects and reports atomic-set serializability violations. The dynamic analysis uses a finite state machine abstraction to represent the progression of an execution through each of the unserializable memory access sequences. If a state machine reaches an accept state, there has been an atomic-set serializability violation. The authors assume that related data are given explicitly by a programmer or expressed in class definitions. All reported errors are with respect to the data relations provided to the analysis.

Von Praun and Gross developed a related technique called *object race detection* [129]. The essential idea is to approximate precise data-race detection by tracking objects, rather than individual memory words. The authors implement a specialized form of Eraser's lockset analysis [119] that manipulates data at the granularity of objects and uses a stricter definition of the locking discipline, permitting only one lock per object. The authors' main focus is, in fact, not the detection of errors involving multiple memory accesses, but rather to improve the performance of standard race detection using a coarse approximation.

MUVI [78] is a technique for automatically inferring which accesses in a program are correlated with one another and using inferred access correlations to identify programming errors. The authors develop a simple definition under which two memory accesses occur *together* – they execute in the same function and there are fewer than a specified number of statements between them (statically). To identify access correlations, the authors analyze the source code, finding sets of accesses that occur together. Next, they make use of a the FPClose [53] itemset mining algorithm to determine which sets of accesses occur frequently. Finally, guided by a set of common programming errors, the authors develop heuristics that use their inferred access sets to find programming errors involving multiple accesses.

Comparison with this work

These techniques are related to ColorSafe, which primarily focuses on multi-variable errors and to Bugaboo and Recon, which address many multi-variable errors. In ColorSafe, we were inspired by Hammer *et al* [55] in our use of serializability analysis modified to reason over grouped data.

There are several key differences between our work and these techniques. ColorSafe *avoids* multi-variable atomicity violations in addition to detecting them. Failure avoidance in ColorSafe is further contrasted with prior work on atomic-set-serializability in Section 8.2.3. Bugaboo and Recon detect a more general class of errors than these techniques. Bugaboo and Recon detect single-variable and multi-variable atomicity and ordering violations; these prior techniques focus on multi-variable atomicity violations. Bugaboo and Recon take an orthogonal approach to detecting multi-variable errors. Bugaboo and Recon do not rely on data-grouping or serializability properties. Bugaboo finds multi-variable errors by finding anomalies in the CACG distinguished by variation in communication context, which abstractly encodes accesses to multiple memory locations. Recon characterizes multi-variable errors errors by including multiple communicating accesses in each reconstruction. MUVI is largely orthogonal to all of our work. MUVI's focus is inferring related data. ColorSafe relies on trivial data-grouping heuristics, or assumes data are grouped *a priori*. Bugaboo and Recon do not need information about grouped data.

8.1.6 Exposing and Reproducing Failures

Much of our debugging work has focused on the problem of starting from a program failure, inferring the error that caused the failure, and providing information to help fix the error. Other prior work has focused on the related but orthogonal goal of *explorative testing* to expose new program failures. These techniques are included in this chapter because of the important link between testing (finding errors) and debugging (fixing errors).

CTrigger [105] is a technique for exposing atomicity violations. CTrigger analyzes program traces. The analysis finds perturbations of the trace that lead to unserializable interleavings of groups of accesses. After finding such perturbations, CTrigger runs the program

repeatedly, inserting delays to try to force the execution to exercise the unserializable interleaving. If the forced interleaving causes a failure, CTrigger reports the likely error and the conditions to reproduce it. If the interleaving does not lead to a failure, CTrigger reports nothing.

CHESS [96] is a directed testing technique. CHESS replaces the thread scheduler used by the program being tested with a special scheduler designed to expose more thread schedules than the default scheduler. The key idea in CHESS is to use directed, iterative search over the space of possible thread interactions. CHESS runs a concurrent program one thread at a time. When a thread reaches a yield point (synchronization, etc) CHESS deschedules it and schedules another thread. After observing such an execution, CHESS attempts to insert a small number of preemptions to perturb the schedule, producing a new schedule. The hope is that perturbing the execution will cause new failures will occur.

Burckhardt *et al* [25] developed a technique similar to CHESS for exploring the space of possible thread schedules of a program to surface new concurrency errors. The authors introduce the notion of *bug depth*. Bug depth is a property of a concurrency bug and is the number of pair-wise event orderings required in an execution for that bug to cause a failure. The authors use an iterative testing strategy based on assigning a priority order to threads and scheduling them according to that priority. The authors use their priority-based scheduler and the notion of bug depth to formally state the probability that an error will be exposed as a failure.

In the same vein, there has been a great deal of work on schemes to record and replay executions [135, 116, 94]. The goal of these techniques is not exposing new failures, but rather recording nondeterminism during live executions. Recorded executions allows failing executions to be precisely reproduced during debugging.

Comparison with this work

These testing techniques are primarily concerned with the first step in dealing with concurrency errors: finding an execution that leads to a failure. In contrast, our debugging work is primarily concerned with the next step: given a failure, help a programmer understand and fix the error that caused it. It is likely that our debugging work would be complemented well by such explorative testing techniques: they can find the failure, and we

can show the programmer how to fix the bug.

Our work on reconstructing program execution fragments bears some similarity to the proposals that report execution schedules from failing executions. Our work differs in the amount of information provided. CHESS and Burckhardt *et al* yield an entire execution schedule, along with a set of "preemption points" required to trigger the failure. While extremely valuable for reproducing a failure, this information is not necessarily helpful for debugging. The reason is that an entire schedule may be billions of operations – clearly too much information for a programmer to digest. The set of preemption points may be useful in showing the programmer approximately where to look to in their program to find the error. However, the preemption points are not guaranteed to occur near the accesses involved in the error – the preemption points need only permit the sequence of instructions that leads to the failure. In contrast, Recon provides a focused sub-sequence of the schedule of execution around a pair of communicating instructions inferred to be related to the failure.

Explorative testing and replay are instrumental in dealing with concurrency errors. They complement our work nicely by exposing new failures and making failure reproducible. Our techniques are able to then point to focused parts of a program likely to be involved in causing those failures.

Another important connection between this area of prior work and the work in this dissertation is to Aviso. Aviso is based on the *Testing-Avoidance Duality*, described in Chapter 7. The testing-avoidance duality is based on the concept of bug depth, proposed in the Burckhardt *et al* work. Characterizing what causes bugs to lead to failures was an instrumental step along the path to developing Aviso, a technique for automatically avoiding those failures.

8.1.7 Statistical Debugging Techniques

There have been several techniques proposed for both sequential and concurrent programs that rely on statistically modeling programs and program executions to help find anomalies and errors.

Daikon [41] is a technique for automatically detecting invariants from program behavior.

The technique works by analyzing execution traces and computes properties of program state (*e.g.*, variables) and of expressions over program state (*e.g.*, $x > 5$). Properties that hold consistently over a trace are reported as invariants. Invariants can be used for a variety of things, such as helping to identify programming errors that lead to invariants being violated, making the detected implicit invariants explicit in the code, and improving the quality of program test suites. Daikon is relevant to this work because it is among the first work combining dynamic analysis and statistical reasoning over program behavior with correctness in mind.

Engler *et al* [40] developed a technique for finding programming errors by looking for *deviant behavior*. Their static analysis builds a statistical model of typestate changes and data- and control-flow patterns that they call their set of "beliefs". Beliefs are based on a set of "templates" formulated by the authors. The strength of a belief is proportional to the frequency with which a pattern is observed. The authors' checker uses the belief model to find deviant behavior – instances of a template not in accordance with strongly held beliefs. The authors tool reports deviant behavior as an error. This technique is not explicitly targeted to concurrency, but is able to cover some concurrency errors (*e.g.*, mutex typestate errors).

Cooperative Bug Isolation [75, 74] (CBI) is a technique for finding bugs in programs that relies on collecting execution properties from deployed software. The technique works by adding lightweight instrumentation to programs that samples state such as variable values and control-flow properties. The system also monitors for failures. Sampled execution properties and failure information are analyzed offline using statistical inference and machine learning to determine which sampled behavior is most likely to be related to a failure. Cooperative Crug Isolation [61] (CCI) is a follow-on to CBI that focuses on finding concurrency errors. CCI uses the CBI approach, but samples events from program executions that are related to concurrent program behavior. The events that CCI adds to CBI are patterns of thread interleavings that correspond to particular concurrency error manifestations (unserializable interleavings, for example). The main novelty of CCI is developing a method for efficiently sampling concurrent behavior.

Comparison with this work

While not strictly focused on concurrency issues, these techniques use statistical and invariant models of software to aid in debugging. Furthermore these techniques are useful for dealing with a variety of failure modes, including some related to concurrency, warranting their discussion in this chapter.

In addition to the use of statistical modeling of program behavior like Bugaboo and Recon, these technique bear additional similarity to our work.

Daikon and the work of Engler *et al* work from the premise that the common case is likely to be correct and that deviations from the common case are more likely to be errors. Bugaboo and Recon rely on the same idea. CBI takes advantage of correlations between the frequency of certain program behaviors, and the occurrence of failures. Bugaboo, Recon, and Aviso employ the same idea: if some behavior tends to occur when some failure also tends to occur, that behavior may be related to that failure.

There are several differences between our work and these techniques.

First, Bugaboo, Recon, and Aviso differ from CBI in that CBI relies heavily on sampling to collect data and the collected data comes from many different program instances. In contrast, our work focuses on the related but different problem of efficiently and continuously collecting information from a single program instance. In Bugaboo, we did so using hardware, which is a contrast to CBI's use of sparse sampling. Sampling and hardware acceleration are likely complementary – our technique could potentially be implemented using sparse sampling (a possible avenue for future work), and CBI may benefit from some hardware support to make its per-node analysis costs lower.

Second, our work uses execution properties related to concurrency. Bugaboo and Recon focus on communication and Aviso focuses on short multi-threaded event traces. By contrast, these prior techniques (except CCI) are not specifically concerned with concurrency. Using such features allows these techniques to cover a wide range of programming errors, including many types of sequential errors. However, the lack of explicit consideration of concurrency errors means that some purely schedule-dependent failures may not be well-handled by these techniques.

CCI focuses on concurrency issues, but, being pattern-based, is limited in generality compared to Bugaboo and Recon. Note, however, that the authors of the CCI work are pri-

marily concerned with the issue of efficiently sampling events in concurrent execution, not on developing new detection criteria for classes of errors. As such, CCI is likely complementary to our work in the same way as CBI.

Aviso is connected to prior techniques on statistical debugging and bug detection because Aviso's mechanism for avoiding failures relies on a statistical model. Aviso's model represents schedule constraints, differing considerably from those used in prior work that represent bug-specific properties or semantic program properties.

8.2 Avoiding Schedule-Dependent Failures

The field of automatically avoiding software failures is somewhat less mature than the field of detecting and debugging errors. This section discusses several important categories of automatic failure avoidance, including foundational work in the area.

8.2.1 Avoiding SC Violations

Ceze's BulkSC [27], Vallejo *et al*'s Kilo-Instruction Processor [32], and Wenisch *et al*'s Store-Wait-Free Processor [132] detect SC violations in order to enforce SC. The goal of these techniques is to make use of *continuous speculation* to reap the performance benefits afforded by relaxed memory semantics, but ensure executions are SC.

To expose performance, these techniques execute programs as coarse-grained speculative blocks. Instruction reorderings are permitted inside blocks, exposing the performance benefits afforded by relaxed memory models. Using scalable store-buffers and disciplined block ordering [132], a modified re-order buffer and load-queue design [32], and Bulk disambiguation of coarse grained speculative regions [28], all three designs prevent SC violations, while at the same time permitting aggressive optimizations allowed by weaker memory models.

Comparison to this work

We showed in Atom-Aid and ColorSafe that systems like these that provide implicit atomicity avoid not just SC violations, but also some atomicity violations. Atom-Aid and ColorSafe dovetail with these prior efforts because they develop a policy for dividing atomic blocks that avoids atomicity violations more effectively than implicit atomicity alone.

8.2.2 Data-Race Exceptions

Goldilocks [39] is a precise race detector that is not based on vector clocks. Instead, Goldilocks maintains a record of the locks held when each memory location was accessed, as well as which threads may access the location at each point during the execution. As the program executes, threads update the contents of these sets based on their synchronization operations. When a memory location is accessed by a thread not in its thread set, or that doesn't hold the proper locks, a race is reported. The Goldilocks algorithm is a modified version of the lockset algorithm [119]. A distinguishing feature of Goldilocks is that it provides fail-stop semantics when data-races occur – the proposed implementation causes the Java Virtual Machine to throw a DataRaceException when Goldilocks detects a race.

DRFx [90] is a memory model designed to provide reasonable semantics for programs containing data-races in the presence of compiler and hardware optimizations. The key idea to DRFx is to use a special compiler to add fences to a program that split it into *regions*. The program is compiled and optimized like normal, except that compiler optimizations introducing speculative memory operations are prohibited. When the program executes, the DRFx hardware monitors region execution and traps whenever two regions are executing concurrently, both access the same memory location and at least one access is a write. DRFx provides a guarantee that if no trap occurs, no violation of SC has occurred and if a trap occurs, then a data-race has occurred (but not necessarily an SC violation). The authors describe hardware and compiler support for implementing DRFx that inserts fences strategically to avoid unbounded buffering requirements during execution and to efficiently monitor for data-races.

The author's own related work on Conflict Exceptions [84] is an execution model that prevents data-races from causing SC violations. Conflict Exceptions works by dividing the execution into spans of code containing no synchronization, called synchronization-free regions (SFRs). Conflict Exceptions shows that if two SFRs overlap and contain memory operations that conflict (*i.e.*, constitute a data-race) then the program may experience an SC violation. In that situation Conflict Exceptions throws an exception, halting the execution. In contrast, if SFRs do not overlap or do not conflict, then no SC violation is

possible, so Conflict Exceptions lets the execution continue. The guarantee provided by Conflict Exceptions is that all exception-free executions are SC and if an exception occurs, then there was a data-race in the execution. Conflict Exceptions uses hardware support to implement its execution model.

Comparison with this work

Goldilocks, DRFx, and Conflict Exceptions are most similar to Atom-Aid, ColorSafe, and Aviso. They key similarity to these prior techniques is that all of these techniques aim to limit the potential harm that can result from data-races. These prior techniques do so by stopping the execution once it is possible that an SC violation has occurred. The work in this dissertation also aims to limit the harm that concurrency bugs can cause. Rather than simply stopping an execution when a failure may have occurred, the work in this dissertation tries to avoid potential failures. Atom-Aid and ColorSafe perturb the execution schedule using transactional memory; Aviso perturbs the schedule using delays.

A major difference between the work in this dissertation and these prior efforts is that these prior efforts aim to provide support for language-level memory models. In contrast, the work in this dissertation aims only to avoid failures, not to furnish guarantees to a language implementation.

8.2.3 Synchronization Synthesis

Prior work on synchronization synthesis is related to Atom-Aid, ColorSafe, and Aviso.

Autolocker [92] is a static approach to avoiding concurrency errors by automatically synthesizing synchronization constraints for atomic regions. AutoLocker assumes a programming model that uses explicit atomic regions for synchronization. The synchronization constraints are implemented by the AutoLocker compiler using locks. The locks that need to be held at each program point are computed using annotations provided by the programmer explaining which lock protects which data. The result of the AutoLocker analysis is a program that is correctly synchronized with respect to the atomicity constraints and mapping from data to locks specified by the programmer.

In the same vein as AutoLocker, Hammer *et al* [55] develop a technique for automatically

synthesizing synchronization constraints based on atomic-set serializability. The authors then show that static analysis can add synchronization to prevent atomic-set serializability violations. The system adds lock-based synchronization to the program. A lock is associated with each atomic set. Before the first access to any variable in the set, the set's lock is acquired. Before returning from a function manipulating a set of variables, the lock is released. The analysis ensures that locks are correctly held to prevent any data-races or atomic-set serializability violations.

Ceze *et al* [26] develop system support for *data-centric synchronization* (DCS). DCS is a method for associating related data with one another by assigning them the same *color*. There is a "critical section" associated with each color. The authors describe a programming model for assigning colors to data and system and architectural support for entering and exiting critical regions and tracking the map from data to their color. A thread enters a color's critical region on its first access to data of that color. The thread in the critical region leaves the critical region when some later "exit" condition is met – *e.g.*, a function return. DCS is closely related to atomic-set serializability, as DCS essentially provides synchronization that ensures atomic-set serializability[1] with respect to an explicitly provided "color scheme" and does so with hardware support. The mapping from data to colors is maintained by the proposed system using a specialized version of the Mondriaan Memory Protection system [133].

Another interesting data-point in this area is ReEnact [111], already discussed in Section 8.1.1. The relevance to our failure avoidance work is that ReEnact has a limited facility to "repair" data-races by matching them against a database of race patterns. ReEnact then uses TLS support to prevent the data-race in the future.

Comparison with this work

There are several key similarities between this work and our work. These techniques and our work both aim to ensure the atomicity and correct ordering of groups of accesses intended by the programmer to be synchronized. Furthermore, like Atom-Aid and ColorSafe, DCS

[1]This is true only if DCS uses transactions. Using a one-lock-per-color scheme enforces a coarser serialization than is required for atomic-set serializability. For example, two threads performing concurrent sequences of read-only accesses to a color would be serialized unnecessarily under the lock-based scheme

uses hardware support to change the execution schedule and prevent failures. ReEnact is similar to Aviso in that it starts from a failure and generates schedule constraints for future executions to prevent failures. The mechanism in ReEnact differs (requiring TLS hardware), but ReEnact and Aviso both aim to enforce learned schedule constraints at run time.

There are also several key differences between our work and these techniques. First, in contrast to these approaches, our technique is not itself a synchronization mechanism, like DCS and the Hammer *et al* work. Instead, Atom-Aid, ColorSafe, and Aviso operate independently of programmer specified synchronization. Our technique is *composable* with synchronization synthesized by AutoLocker, DCS or Hammer *et al* and can provide benefit, even in cases where the programmer has incorrectly specified synchronization constraints.

Second, because Atom-Aid and ColorSafe are not a synchronization model and so do not need to provide strong correctness properties, they can be *less conservative* than these prior techniques. DCS serializes conflicting groups of accesses to a color by different threads. DCS conservatively starts transactions on any access to colored data for which a transaction is not already in progress, to keep the colored data consistent. Hammer *et al* inserts synchronization to prevent all *possible* atomic-set serializability violations – many never actually occur in practice, but are synchronized away nonetheless. Our technique only starts a transaction on an access to data previously involved in an unserializable interleaving and for which there is not already a transaction in progress.

These techniques are related to Aviso in that it empirically derives synchronization constraints. However, Aviso solves a fundamentally different problem than these techniques. These techniques focus on generating synchronization constraints for the entire program, starting from a program and a correctness specification. Aviso synthesizes schedule constraints involving specific parts of a program, after observing a failure. Furthermore, the constraints generated by these prior techniques are intended to be used for the lifetime of software. Aviso is intended to be *triage*, preventing failures until a patch can be released.

8.2.4 Avoiding Atomicity Violations

Techniques for avoiding atomicity violations are related to Atom-Aid, ColorSafe, and Aviso.

Isolator [113] is a technique for ensuring the atomicity and isolation of accesses by one thread in a lock-based critical region, even in the presence of accesses to the same variables in another thread outside a critical region. When a thread enters a critical region, it copies the pages containing the data to be accessed and sets the page protections on the original pages to trigger a fault if another thread accesses them. The thread in the critical region manipulates the copied pages during the critical region. If another thread accesses the original pages, a protection fault occurs and the thread is made to wait by the system for a fixed period of time. During that time, the thread in the critical region can finish its accesses. After finishing its critical region, the thread copies the modified pages back to the original pages and removes the page protections to prevent subsequent faults. The technique forces racy accesses that race with accesses in a critical region in another thread to occur after the critical region rather than during the critical region.

ToleRace [114] is another technique for ensuring that accesses made in a critical region are atomic and isolated, in spite of racy accesses made concurrently with the critical region. The authors first provide a characterization of the races they detect and tolerate, which they call *asymmetric races*. Using their characterization, they describe their technique, which, like Isolator, changes program behavior in critical regions. They key idea is to make local copies of data upon entering a critical region (*i.e.*, acquiring a lock). All accesses to data made during a critical region are made to the local copies and upon leaving the critical region, modified local copies of the data are copied back to their original locations. The authors describe an implementation based on binary instrumentation that implements copying and deals with a variety of synchronization types.

Kivati [30] is a static analysis and dynamic run time monitor for avoiding atomicity violations. Kivati works by first using static analysis to infer which pairs of accesses may need to be atomic. Kivati's analysis looks at each function and conservatively approximates which data might be shared. The analysis then looks at the control-flow graph for the function and finds pairs of accesses to the same shared variable that may be consecutive. The analysis marks such pairs with inferred atomicity constraints. At runtime, before the first of a pair of accesses executes, Kivati sets a hardware watchpoint on the data being accessed. If any thread except the thread executing the pair accesses the data, the

watchpoint is tripped indicating an atomicity violation would have occurred, were it not for the watchpoint trap. The access that caused the trap is delayed and replayed later when it does not cause an atomicity violation. To minimize the frequency of watchpoint traps, Kivati uses specialized serializability analysis to strategically use watchpoints only where unserializable interleavings may occur.

Comparison with this work

These techniques are closely related to Atom-Aid, ColorSafe, and Aviso. At a high level, these techniques and our work are related in that all aim to prevent atomicity violations. These techniques focus on a particular class of atomicity violation errors, included in the class of atomicity violations dealt with by our techniques.

Despite these similarities, our techniques differ in several ways from Isolator, ToleRace, and Kivati.

First, Atom-Aid and ColorSafe target a broader class of atomicity violations than Isolator and ToleRace. Aviso targets a broad class of concurrency failures that includes atomicity violations. In order for Isolator and ToleRace to prevent failures, at least some of the code involved in a failure must be correctly synchronized (*i.e.*, in a critical region). Relying on existing synchronization facilitates the Isolator and ToleRace analysis and reduces the set of program points that must be analyzed (and potentially transformed). The simplicity leads to good performance, but makes the techniques useful in fewer situations than our techniques.

Second, the mechanisms underlying our techniques operate completely orthogonally to these techniques. Our techniques apply to programs after they have been compiled, manipulating memory accesses at the level of hardware in the case of Atom-Aid and ColorSafe. With such a low level of abstraction from the execution, our techniques are likely to compose well with higher level techniques like Isolator and ToleRace.

Third, these techniques differ from our work in their approach to inferring which accesses should be atomic. As with our work, these techniques use heuristics for determining which sequences of accesses should be made atomic. Isolator and ToleRace assume accesses in lock-based critical regions should be atomic with respect to all other accesses, not just those protected by the same lock. Kivati assumes consecutive accesses in the same function

to the same program-level variable should be atomic. Atom-Aid and ColorSafe assume dynamic sequences of accesses beginning with an access to suspect data should be atomic. Aviso is more focused, empirically determining which access sequences should be atomic.

8.2.5 Online Patching and Patch Synthesis

Wu *et al* develop Loom [134], a technique that allows programmers to write restricted patches that prohibit certain thread schedules. The system interprets the patches and enforces multi-threaded schedule constraints. The system is implemented with a focus on availability and as such, patches can be applied without stopping the program being executed. The key to the system's availability guarantees lies in a safety analysis that determines when new synchronization instrumentation code can be added without introducing an error.

Jin *et al* [59] develop AFix, a technique that automatically patches programming errors to eliminate concurrency bugs detected using prior automatic error detection techniques. The key idea is to use error reports for single-variable atomicity violation bugs with a static analysis that generates synchronization constraints. The generated synchronization constraints prevent the atomicity-violating interleaving reported by the bug detection tool. The system's analysis makes special considerations to ensure that added synchronization operations do not cause deadlocks and tries to find efficient patches.

CFix [62], Jin *et al*'s follow-on work to AFix, takes a more general approach to the problem of automatically patching concurrency bugs. CFix uses bug detection tools as a "subroutine". Using those tools, CFix finds bugs and tries to generate synchronization code to enforce mutual exclusion and ordering constraints that prevent those bugs from manifesting as failures. The key difficulties that CFix overcomes are the potential for low performance introduced by the automated patches and the potential for deadlock introduced by inserted synchronization code.

Perkins *et al* develop ClearView [107], a system for general automatic patch generation. The technique works by building a model of observed invariant behavior over a period of non-failing execution. When a failure is detected, the system finds a set of invariants that

were violated. Based on these invariants, the system generates a set of candidate patches. Candidate patches are empirically evaluated. Those that do nothing or negatively affect future executions (*i.e.*, cause crashes) are discarded. Those that are effective are left applied to the application, preventing future failures.

Comparison with this work

Loom, AFix, and ClearView are related to Aviso. The main similarities are that like these systems, Aviso patches software automatically and without requiring programmers to go through the trouble of rewriting programs.

There are several distinctions between these techniques and Aviso, however.

Loom is outpaced by AFix, ClearView, and Aviso in its degree of automation. Loom requires programmers to provide patches that fix bugs to the system. In order to provide a patch, a programmer must understand the error behavior adequately to describe how to prevent it. Often the most difficult part of dealing with a concurrency error is developing such an understanding. As a result, Loom is mainly effective only for errors that are already well-understood. In contrast, AFix, ClearView, and Aviso all focus on deriving the "understanding" of the bugs automatically, without the participation of the programmer.

AFix focuses on a narrow stripe of concurrency defects – single-variable atomicity violations. These are an important category of errors, but restricting focus to only this category of errors limits the applicability of AFix. Furthermore, AFix is also limited to the class of atomicity violations that can be detected by the detection technique that underlies AFix (in the reference implementation, CTrigger [105]). CFix is more general than AFix. While AFix relies on CTrigger to identify atomicity violation root causes, CFix abstracts the bug detection tool, allowing any to be used. CFix's generality is dictated only by the generality of the underlying detection tool.

In contrast to AFix, Aviso is more general. Aviso can generate schedule constraints that prevent atomicity violations and ordering violations involving one or many variables. The generality of CFix and Aviso are hard to compare because Aviso relies on observing deployed execution and CFix relies on good bug detection tools. On the one hand, detection tools are sometimes limited in the bugs they can find, which could make CFix less capable than Aviso, which can deal with any schedule-dependent failure that shows up in production.

On the other hand, CFix's use of detection tools may help expose unlikely failures earlier than they might show up in production, allowing it to attack more bugs more quickly than Aviso.

There are many mechanical differences in the way Aviso, CFix, and AFix operate. AFix and CFix are more labor-intensive than Aviso. AFix and CFix require a programmer to use the underlying bug detection tools, providing appropriate test inputs and monitoring test executions. In contrast, Aviso only requires a system to run in production and is fully automated from end to end, after software is deployed. Aviso is a *dynamic* mechanism, avoiding failures by applying schedule constraint in its runtime, during the execution. CFix and AFix are static techniques that use their analysis results to modify the program's code.

ClearView is invariant-guided, making use of Daikon [41] to provide invariants. Daikon is not explicitly tuned to focus on concurrency errors, so there are likely important concurrency-related failures that ClearView would not be able to handle. In contrast, Aviso is explicitly focused on concurrency-related failures. While different in their focus, ideas in ClearView and ideas in Aviso may be complementary to one another.

8.2.6 Cooperative Failure Avoidance

There are some important instances of cooperative failure avoidance, related to Aviso, that show up in prior work.

Exterminator [100] is a technique for automatically avoiding failures due to heap memory errors. The key idea is to exploit memory space randomization to find where errors may occur. Exterminator creates "online patches" to avoid those errors in future executions. Different program instances can also cooperatively share and combine patches.

Communix [64] is a technique that automatically avoids deadlock failures. The key idea in Communix is that many program instances use the Dimmunix [65] algorithm to find patterns of lock acquires that lead to deadlock. Dimmunix can then prescribe, for each of these, how to avoid that deadlock. Then Communix provides a mechanism that different software instances can use to share and compare deadlock avoidance strategies. Communix is, therefore, a collaborative mechanism for improving the precision and coverage

of Dimmunix.

Comparison with this work

Exterminator and Communix are most similar to Aviso. Like Aviso, Exterminator and Communix leverage large communities of software instances. A key difference is that Aviso generates a large set of hypothetical schedule perturbations, each of which might avoid a failure, and distributes them to participating instances. Exterminator largely relies on randomization and variation within instances to find effective fixes. Then, Exterminator uses instance collaboration to combine patches. Communix is like Exterminator in that patches are determined by single instances, using the Dimmunix algorithm. Similarly to Aviso, patches can be shared between instances in Communix, like Aviso's schedule constraints.

A major difference between these systems and Aviso is their applicability. Exterminator focuses on a subset of memory errors. Communix targets bugs that can be detected by Dimmunix. In contrast, Aviso targets general schedule-dependent failures. Deadlocks that Dimmunix can detect are schedule-dependent, so they are handled by Aviso. Aviso does not explicitly deal with memory errors and Exterminator does not explicitly address concurrency, making it difficult to assess the overlap in their applicability.

8.2.7 Schedule-Memoization-Based Failure Avoidance

PSets [139] is a multiprocessor design that avoids failures by using special system support for schedule *memoization*. PSets works by starting with a training phase. During the training phase the system observes many non-failing execution schedules. In each non-failing execution, the system tracks the *predecessor* of each memory access. The predecessor of an access is an access that preceded that accesses, accessed the same memory location, and executed in a different thread. After training, PSets uses the set of predecessors for each access to constrain the execution schedule. If an access is about to occur and its predecessor is an access that was never its predecessor during training, the access is delayed and in some cases the predecessor is rolled back. The goal in doing so is to force the schedule back to an observed schedule that was seen during training and so is more likely to have been tested.

Tern [33], like PSets, works by memoizing correct executions. However, unlike PSets,

Tern's goal is slightly different. Tern aims to provide *stable deterministic execution*. While Tern does not provide determinism in the sense of other approaches to deterministic execution [37, 15, 101], it ensures that similar inputs execute under the same concurrent schedule. The way that Tern achieves this goal is by memoizing schedules. Before each scheduling decision is made, a data-structure containing previously memoized schedules is queried. If at that point in some prior execution a scheduling decision was made and memoized, the same decision can be made again, without risk of a concurrency failure.

Dimmunix [65] is a system that can avoid some deadlock bugs. Dimmunix works by monitoring the order of lock acquires during program executions. When a system reaches a likely deadlock state, Dimmunix analyzes the order of lock acquires from that execution. The analysis reports sequences of lock acquires that are likely to have been responsible for causing that deadlock. Dimmunix can use those sequences in future executions to avoid experiencing those same deadlocks again.

Comparison with this work

Schedule memoization techniques are, in a sense, dual to the technique we use to avoid failures in Aviso. Schedule memoization works by observing execution schedules that do not lead to failures and adhering to those schedules in future executions using explicit scheduling constraints. In contrast, Aviso works by observing execution schedules that do lead to failures and avoiding those schedules in future executions using explicit scheduling constraints.

We have not evaluated this, but Aviso is likely to be more flexible and adaptive to new situations than PSets, because Aviso does not use a fixed set of memoized schedules. Instead, when a failure occurs, Aviso can learn new failure-avoiding constraints on the fly and add them to the set of constraints avoiding failures in the application.

Aviso is related to Dimmunix. Both reason about event sequences that are likely to have led to a failure. Both systems also use those event sequences to prevent failures in future executions. Aviso differs from Dimmunix in that it targets a more general class of bugs than just deadlocks, like Dimmunix. Unlike Aviso, Dimmunix limits its search for constraints on future execution schedules to information from a single execution. In contrast, Aviso incorporates information from many failing and non-failing execution schedules into its

schedule constraint selection model.

8.2.8 Determinism

Deterministic execution is a technique for executing concurrent programs and adhering to a single thread schedule for each execution on the same input. The key idea is to eliminate scheduling decisions that occur during an execution and have an impact on the order in which events execute. There have been a large number of recent efforts in the area of deterministic multi-threaded execution. Some techniques rely on hardware [37], some on compiler and runtime support [101, 15], some on system support [16, 77], and some on determinism guarantees provided by the programming language [22]. Providing a full survey of these techniques is outside the scope of this dissertation, but there are several things worth mentioning about deterministic execution to relate it to this work.

Comparison with this work

Deterministic multi-threaded execution is most related Aviso because deterministic systems and Aviso both restrict the space of possible thread schedules. Determinism is more generally related to our work because it was developed to improve the state of concurrent programming, debugging, and reliability.

There are several main differences between deterministic multithreading work and the work in this dissertation. First, determinism does not necessarily avoid failures. It may, if a non-failing schedule is executed. However, failing schedules may be executed by deterministic systems as well. Deterministic schedule constraints aim to reduce the space of possible schedules, not to avoid schedules that lead to failures. Aviso focuses on avoiding failures, without the broader aim to reduce the space of execution schedules in general.

Second, our work and determinism both improve the state of concurrent program debugging. However, our approach differs from the deterministic approach in a fundamental way: the debugging benefit of determinism is mainly that failures are reproducible. Bugaboo and Recon do not address reproducibility. They focus on rooting out focused portions of an execution related to a failure.

Third, it is possible that determinism simplifies testing concurrent software [37, 15, 101].

Testing is not addressed by our work.

Chapter 9

CONCLUSIONS

The pervasion of parallel computer architectures and inherently concurrent applications necessitates concurrent and parallel software. Ensuring the correctness and reliability of concurrent and parallel software is a more complex task than doing so for sequential programs. This complexity opens the door for many types of concurrency bugs. Concurrency bugs are very difficult to find, diagnose, and fix. Concurrency bugs that go unnoticed can cause schedule-dependent failures in production systems, degrading the reliability of those systems. Eliminating the barriers to creating correct, reliable concurrent programs is a challenge of critical importance to computer science and the world. The work in this dissertation aims to reduce those barriers.

The thesis of this dissertation is that system and architecture support can simplify concurrency bug debugging and can enable systems to automatically avoid schedule-dependent failures despite latent concurrency bugs. Chapters 2–7 described several designs that demonstrate this thesis. Each chapter's "Conclusions, Insights, and Opportunities" discusses the specific conclusions, limitations, and future directions of the work in that chapter. In addition to those, there are some cross-cutting themes worth mentioning here.

9.1 Cross-cutting Themes

There are several important cross-cutting themes in this work.

9.1.1 Architecture and System Support Across the System Stack

One major theme throughout the work in this dissertation is the application of system and architecture support through the layers of the system stack – from hardware, to low-level software, to the distributed system level, and to the application level. Developing support

across the stack facilitates looking at a problem through the right "lens of abstraction" and balancing important trade-offs like performance and precision.

Examples of this theme abound: Bugaboo uses hardware support to collect information for a dynamic analysis that is exposed to a programmer through a software engineering tool. Atom-Aid and ColorSafe encode a dynamic analysis in hardware that shares an interface with the synchronization mechanism in low-level software. Atom-Aid and ColorSafe's debugging facilities further exposes information up the stack to facilitate debugging. Aviso applies compiler, runtime library, and distributed system support to avoid failures.

9.1.2 Exploiting Execution Schedule Variation

The amount of variation in the multi-threaded execution schedule is one of the key challenges to correctly writing shared-memory multi-threaded programs. The work in this dissertation overcomes the challenge posed by schedule variability by *directly taking advantage of it.* Atom-Aid, ColorSafe, and Aviso capitalize on schedule variation by perturbing schedules that would lead to failures so that different, non-failing schedules execute instead. Atom-Aid and ColorSafe vary the execution schedule by selectively applying atomicity. Aviso varies the execution schedule by directly manipulating the order of select pairs of operations. Bugaboo and Recon also take advantage of execution schedule variation. The property these techniques exploit is that some parts of the execution schedule vary more than others. By observing a diversity of varied execution schedules, Bugaboo and Recon establish what does and does not vary. Contrasting invariant parts of an execution schedule with variant parts helps Bugaboo and Recon reveal parts of a program related to failures being debugged.

9.1.3 Leveraging Collective Behavior and Statistical Modeling

Several techniques in this dissertation use information about the *collective* behavior of systems to build useful models of those systems' behavior. Bugaboo and Recon aggregate information from many program executions to find behavior invariant in failing and non-failing executions. Aviso relies on the cooperation of a population of program instances to find effective schedule constraints. Incorporating information from many program execu-

tions facilitates building statistical models of system behavior. The work in this dissertation uses such statistical models, which improve as the amount of data collected increases.

9.1.4 Utility for System Lifetime

The mechanisms described in this dissertation are useful for a system's lifetime: during development and in production. Atom-Aid, ColorSafe, and Aviso avoid schedule-dependent failures, which provides benefit to production systems, and also produce reports of likely bugs to programmers, which are useful for development. Bugaboo and Recon are useful for development-time debugging. When combined with hardware support for collecting context-aware communication graphs, these techniques become useful after deployment by furnishing developers with graphs from failing production runs.

9.2 Final Thoughts

This dissertation showed that with system and architecture support the challenges associated with complex concurrent and parallel software can be made manageable. Support for debugging and failure avoidance improves the quality of software and its eventual reliability in production. The work in this dissertation illustrates several concrete instances of system and architecture support that accomplish this goal. This work, combined with related and future work on testing, bug detection, and new programming models, moves the software world toward correct, reliable concurrency and parallelism, overcoming its fundamental complexities.

BIBLIOGRAPHY

[1] Advanced synchronization facility proposed architectural specification. http://developer.amd.com/wordpress/media/2013/02/45432-ASF_Spec_2.1.pdf.

[2] The go programming language specification. http://golang.org/ref/spec.

[3] Intel architecture instruction set extensions programming reference. http://software.intel.com/sites/default/files/m/9/2/3/41604.

[4] International standard - programming languages - c. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[5] The open group base specifications issue 6 ieee std 1003.1, 2004 edition. http://pubs.opengroup.org/onlinepubs/007904975/basedefs/pthread.h.html.

[6] The opencl specification document revision 19. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[7] Openmp application program interface. http://www.openmp.org/mp-documents/spec25.pdf.

[8] Parallel programming and computing platform cuda nvidia. http://www.nvidia.com/object/cudahomenew.html.

[9] The sparc architecture manual. http://www.sparc.com/standards/SPARCV9.pdf.

[10] U.s.-canada power system outage task force final report on the august 14, 2003 blackout in the united stats and canada: Causes and recommendations. https://reports.energy.gov/BlackoutFinal-Web.pdf.

[11] Summary of the amazon ec2 and amazon rds service disruption in the us east region. http://aws.amazon.com/message/65648/, April 2011.

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

[13] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th annual international symposium on Computer architecture*, ISCA '91, pages 234–243, New York, NY, USA, 1991. ACM.

[14] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *NDDL/VVEIS*, pages 82–93, 2003.

[15] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[16] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM.

[17] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[18] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[19] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[20] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[21] Colin Blundell, E. Lewis, and Milo Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking*, 2005.

[22] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems*

languages and applications, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[23] Hans-J. Boehm. How to miscompile programs with "benign" data races. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[24] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[25] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 167–178, New York, NY, USA, 2010. ACM.

[26] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 133–144, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 278–289, New York, NY, USA, 2007. ACM.

[28] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[29] A. Chang and M. F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions Computer Systems*, February 1988.

[30] Lee Chew and David Lie. Kivati: fast detection and prevention of atomicity violations. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 307–320, New York, NY, USA, 2010. ACM.

[31] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[32] Adrián Cristal, Oliverio J. Santana, Mateo Valero, and José F. Martínez. Toward kilo-instruction processors. *ACM Trans. Archit. Code Optim.*, 1:389–417, December 2004.

[33] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[34] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM.

[35] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfets with very small physical dimensions. 9, October 1974.

[36] Memcached Developers. Issue 127: incr/decr operations are not thread safe. http://code.google.com/p/memcached/issues/detail?id=127.

[37] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM.

[38] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. Ifrit: interference-free regions for dynamic data-race detection. *SIGPLAN Not.*, 47(10):467–484, October 2012.

[39] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 245–255, New York, NY, USA, 2007. ACM.

[40] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[41] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[42] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[43] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):5666.

[44] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71:89–109, April 2008.

[45] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, November 2010.

[46] Cormac Flanagan and Stephen N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '10, pages 1–8, New York, NY, USA, 2010. ACM.

[47] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30:20:1–20:53, August 2008.

[48] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 293–303, New York, NY, USA, 2008. ACM.

[49] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 338–349, New York, NY, USA, 2003. ACM.

[50] Qi Gao, Feng Qin, and Dhabaleswar K. Panda. Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 15:1–15:12, New York, NY, USA, 2007. ACM.

[51] Kourosh Gharachorloo and Phillip B. Gibbons. Detecting violations of sequential consistency. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, SPAA '91, pages 316–326, 1991.

[52] J. R. Goodman. Cache consistency and sequential consistency. Technical Report no. 61, SCI Committee, March 1989.

[53] G Grahne and J Zhu. Efficiently using prefix-trees in mining frequent itemsets. 2003.

[54] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[55] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 231–240, New York, NY, USA, 2008. ACM.

[56] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[57] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17:549–557, October 1974.

[58] Robert Hood, Ken Kennedy, and John Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 74–81, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[59] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 389–400, New York, NY, USA, 2011. ACM.

[60] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation. In *OOPSLA*, 2010.

[61] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 241–255, New York, NY, USA, 2010. ACM.

[62] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 221–236, Berkeley, CA, USA, 2012. USENIX Association.

[63] P. Joshi and K. Sen. Predictive typestate checking of multithreaded java programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 288–296, Washington, DC, USA, 2008. IEEE Computer Society.

[64] Horatiu Jula, Pinar Tozun, and George Candea. Communix: A framework for collaborative deadlock immunity. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 181–188, Washington, DC, USA, 2011. IEEE Computer Society.

[65] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 295–308, Berkeley, CA, USA, 2008. USENIX Association.

[66] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, New York, NY, USA, 2012. ACM.

[67] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA*, 2002.

[68] Igor Kononenko. Estimating attributes: Analysis and extensions of relief. In *European Conference on Machine Learning*, 1994.

[69] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 353–367, New York, NY, USA, 2011. ACM.

[70] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[71] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28:690–691, September 1979.

[72] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23:105–117, February 1980.

[73] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[74] Ben Liblit Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Public deployment of cooperative bug isolation. In *In Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS 04)*, pages 57–62. ACM Press, 2004.

[75] Benjamin Robert Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, December 2004.

[76] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[77] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[78] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 103–116, New York, NY, USA, 2007. ACM.

[79] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[80] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[81] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 553–563, New York, NY, USA, 2009. ACM.

[82] Brandon Lucia and Luis Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 39–50, New York, NY, USA, 2013. ACM.

[83] Brandon Lucia, Luis Ceze, and Karin Strauss. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 222–233, New York, NY, USA, 2010. ACM.

[84] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 210–221, New York, NY, USA, 2010. ACM.

[85] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

[86] Brandon Lucia, Benjamin P. Wood, and Luis Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 378–388, New York, NY, USA, 2011. ACM.

[87] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[88] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[89] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM.

[90] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. Drfx: a simple and efficient memory model for concurrent programming languages. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 351–362, New York, NY, USA, 2010. ACM.

[91] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[92] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 346–358, New York, NY, USA, 2006. ACM.

[93] Sang L. Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, pages 235–244, New York, NY, USA, 1991. ACM.

[94] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.

[95] Gordon E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[96] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[97] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 287–297, Washington, DC, USA, 2010. IEEE Computer Society.

[98] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[99] NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing. `http://www.nist.gov/director/prog-ofc/report02-3.pdf`, 2002.

[100] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Commun. ACM*, 51(12):87–95, December 2008.

[101] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, March 2009.

[102] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[103] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 135–145, New York, NY, USA, 2008. ACM.

[104] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 245–254, New York, NY, USA, 2010. ACM.

[105] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. *SIGPLAN Not.*, 44(3):25–36, March 2009.

[106] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM.

[107] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[108] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 179–190, New York, NY, USA, 2003. ACM.

[109] Charles Price. *MIPS IV Instruction Set, Revision 3.2*. MIPS Technologies, Mountain View, CA, September 1995.

[110] Milos Prvulovic. Cord: Cost-effective and nearly overhead-free order recording and data race detection. In *Proceedings of the 12th symposium on High-Performance Computer Architecture*, pages 316–326, 2006.

[111] Milos Prvulovic and Josep Torrellas. Reenact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 110–121, New York, NY, USA, 2003. ACM.

[112] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 111–122, Washington, DC, USA, 2002. IEEE Computer Society.

[113] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 181–192, New York, NY, USA, 2009. ACM.

[114] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. In *IEEE Transactions on Computers*, 2011.

[115] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[116] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999.

[117] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society.

[118] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPOPP*, pages 83–94, 2005.

[119] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, October 1997.

[120] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53:89–97, July 2010.

[121] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do i use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 160–174, New York, NY, USA, 2010. ACM.

[122] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 8–8, New York, NY, USA, 2001. ACM.

[123] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *International Symposium on Computer Architecture*, 2002.

[124] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[125] Enrique Vallejo, Marco Galluzzi, Adrin Cristal, O Vallejo, Ramn Beivide, Per Stenstrm, James E. Smith, Mateo Valero, and Grupo De Arquitectura De Computadores. Implementing kilo-instruction multiprocessors. In *In Proceedings of the 2005 IEEE International Conference on Pervasive Services*, 2005.

[126] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. *SIGPLAN Not.*, 41(1):334–345, January 2006.

[127] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM.

[128] Haris Volos, Andres Jaan Tack, Michael M. Swift, and Shan Lu. Applying transactional memory to concurrency bugs. *SIGARCH Comput. Archit. News*, 40(1):211–222, March 2012.

[129] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 70–82, New York, NY, USA, 2001. ACM.

[130] Liqiang Wang and Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPOPP*, pages 61–71, 2005.

[131] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A practical fpga-based framework for novel cmp research. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, FPGA '07, pages 116–125, New York, NY, USA, 2007. ACM.

[132] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 266–277, New York, NY, USA, 2007. ACM.

[133] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. *SIGARCH Comput. Archit. News*, 30(5):304–316, October 2002.

[134] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing races in live applications with execution filters. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[135] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135, New York, NY, USA, 2003. ACM.

[136] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, June 2005.

[137] Min Xu, Mark D. Hill, and Rastislav Bodik. A regulated transitive reduction (rtr) for longer memory race recording. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 49–60, New York, NY, USA, 2006. ACM.

[138] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.

[139] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. *SIGARCH Comput. Archit. News*, 37(3):325–336, June 2009.

[140] Jie Yu and Satish Narayanasamy. Tolerating concurrency bugs using transactions as lifeguards. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 263–274, Washington, DC, USA, 2010. IEEE Computer Society.

[141] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM.

[142] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.

[143] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. Conseq: detecting concurrency bugs through sequential errors. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 251–264, New York, NY, USA, 2011. ACM.

[144] Wei Zhang, Chong Sun, and Shan Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 179–192, New York, NY, USA, 2010. ACM.

[145] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 121–132, Washington, DC, USA, 2007. IEEE Computer Society.