Efficient, Practical Dynamic Program Analyses for Concurrency Correctness

Dissertation

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

By

Man Cao, B.E., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2017

Dissertation Committee:

Michael D. Bond, Advisor
Atanas Rountev
Feng Qin
Abstract

Shared-memory parallel programs are notoriously difficult to be both scalable and correct. One of the most problematic concurrency bugs is data race. Data races are difficult to avoid, find, fix, reproduce, and eliminate. A fundamental problem is that language and hardware memory models provide few or no guarantees for executions containing data races. Researchers have developed various program analyses and runtime tools for concurrency correctness properties. Examples includes data race detectors, multithreaded record & replay, transactional memory, and enforcement of stronger memory models. However, in the presence of data races, many of these tools suffer from limitations that impede their widespread use.

The first challenge in handling racy executions is the high overhead for tracking (i.e., detect or control) cross-thread dependences, which is a necessary requirement to ensure the soundness of many analyses. The second limitation is that existing work has not covered the full range of possible behaviors for racy executions in weak memory models. This thesis explores several efficient and practical dynamic program analyses that aim to overcome these two key limitations, advancing the state of the art for detecting, enforcing, and exposing issues related to concurrency correctness.

We present hybrid tracking and RegPlay to address the first challenge. Hybrid tracking is a generalized framework for tracking dependences, which hybridizes pessimistic and optimistic tracking in order to get the best of both world. We build hybrid-tracking-based
versions of a dependence recorder and a region serializability enforcer to demonstrate the usefulness of hybrid tracking. RegPlay shows an analysis-specific way of optimizing dependence tracking in the context of multithreaded record & replay. RegPlay avoids recording read–write dependences and many transitively implied dependences in order to reduce run-time overhead, and enforces replay determinism by detecting and resolving violations of read–write dependences. Experiments show that hybrid tracking enables runtime support to overcome the performance limitations of both pessimistic and optimistic tracking alone, and RegPlay records much fewer dependences than existing approach while preserves replay determinism.

To address the second limitation, we introduce *prescient memory* (PM), a novel dynamic analysis that exposes behaviors due to future values—a value written by a store that executes after the load that uses the value. A load could return a future value in a racy execution in weak memory models, but existing analyses fail to expose such behaviors. PM speculatively returns a future value at a program load, and tries to validate the speculative value at a later store. PM applies a novel approach that profiles future values and guides execution to increase the chances of successfully validating future values in real application executions. Experiments shows that PM uncovers a few previously unknown behaviors due to future values in real applications.

Overall, this thesis presents three approaches that overcome limitations of existing work on concurrency correctness. These approaches advance the state of the art in terms of performance, coverage, capability, and practicality. The proposed approaches will improve the efficiency and adoption of dynamic analyses and runtime support for concurrency correctness, which will significantly strengthen software reliability and increase productivity of software development in the years to come.
Dedicated to my family
Acknowledgments

I am deeply grateful to my advisor, Michael Bond, for all his invaluable help, guidance and support during my doctorate study. I have learned a lot from Mike, not only in the academic aspects of technical expertise and methodology of critical thinking and general problem solving, but also in his wholesome attitude and habits for work and life. I am especially thankful for the travel and career development opportunities that Mike offered. Working with Mike is an exceptional experience in my life, which has shaped me into a competent researcher and engineer, laying an indispensable foundation for my future career and life.

I had wonderful experience of interacting and collaborating with professors and researchers inside and outside OSU. I had countless discussions with Milind Kulkarni, who provided insightful feedback and valuable advice on various projects, drafts and presentations. Benjamin Wood has offered inspirational ideas and made significant contributions to several projects, some of which are not included in this dissertation. I thank Atanas Rountev, Feng Qin, and P. Sadayappan to be on my dissertation and candidacy committees. They have taught me a lot about compilers and program analyses, systems, and high performance computing, respectively. Arnab Nandi has introduced me to the fascinating area of data mining and analytics, and helped to turn a class project into a publication. I thank Tim Harris, Doug Lea, Hans Boehm, Harry Xu, Joe Devietti, Xipeng Shen, Brian Demsky, and Olatunji Ruwase for valuable suggestions, reviews and discussions.
The graduate students of the PLaSS group have offered remarkable help and contributions to my research. I am fortunate to have the opportunity to work with Aritra Sengupta, Minjia Zhang, Swarnendu Biswas, Jipeng Huang, Meisam Fathi-Salmi, Jake Roemer, and Rui Zhang.

I had two amazing internships at Google working on projects closely related to my area. Such experience strengthened my engineering skills and positively influenced my research, establishing a brilliant start to my career path. I am grateful to receive exceptional advice and mentoring from Jungwoo Ha, Mathieu Chartier, Chuck Rasbold, Jean Christophe Beyler, Jeremy Manson, Ian Rogers, Eddie Aftandilian, Duncan Hanson, Hiroshi Yamauchi, Andreas Gampe, Hans Boehm, Nicolas Geoffray, and Brian Carlstrom.

My endeavor towards PhD would not be as memorable and pleasant without the fun and support from friends. I would like to express my gratitude to the Reese family for their kind hosting and continued help that makes Columbus feel like my second home. I am thankful for the support from numerous friends and colleagues: Bingyu Xia, Yang Zhang, Lilong Jiang, Jitong Chen, Yue Qiao, Nan Deng, Ziqi Huang, Linchuan Chen, Dachuan Huang, Shengqian Yang, Dacong Yan, Yan Wang, Haowei Wu, Hailong Zhang, Mingzhe Li, Miaojun Yao, Hang Guo, Yu Fan, Zhe Dong, Peiyuan Teng, Yiran Zhang, and Dongdong Zhang.

Finally, I am forever indebted to my parents for their constant support and unconditional love and sacrifice. I am also grateful to my older sister for her care and assistance throughout my life. The persistent encouragement, understanding and support from my family are fundamental to the achievements I have today.
Vita

May 2017 ........................................ Ph.D.,
Computer Science and Engineering,
The Ohio State University, USA.

August 2016 ....................................... M.S.,
Computer Science and Engineering,
The Ohio State University, USA.

May 2009 ........................................ B.E.,
Computer Science and Technology,
Zhejiang University, China.

Publications

Research Publications


**Fields of Study**

Major Field: Computer Science and Engineering

Studies in:
- Programming Languages and Software Systems
- Data Mining
- High Performance Computing

Prof. Michael D. Bond
Prof. Arnab Nandi
Prof. P. Sadayappan
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiv</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview and Outline</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Contributions and Impact</td>
<td>5</td>
</tr>
<tr>
<td>2. Background</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Data Race</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Memory Models</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Implication on Analyses and Runtime Support</td>
<td>11</td>
</tr>
<tr>
<td>3. Hybrid Tracking: Combining Pessimistic and Optimistic Tracking of Cross-Thread Dependences</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Problem and Motivation</td>
<td>14</td>
</tr>
<tr>
<td>3.1.1 Pessimistic Tracking</td>
<td>17</td>
</tr>
<tr>
<td>3.1.2 Optimistic Tracking</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Hybrid State Model</td>
<td>22</td>
</tr>
<tr>
<td>3.2.1 The Pessimistic–Optimistic Mismatch</td>
<td>22</td>
</tr>
</tbody>
</table>
4. RegPlay: Efficient, Software-Only Multithreaded Record & Replay ............ 61

4.1 Background and Motivation .............................................. 61
4.2 Design ................................................................. 67
  4.2.1 Overview ......................................................... 67
  4.2.2 Common Data Structures ...................................... 70
  4.2.3 Record .......................................................... 72
  4.2.4 Replay .......................................................... 77
  4.2.5 Insights and Rationale ........................................ 81
  4.2.6 Correctness of Algorithm 8 .................................... 85
  4.2.7 Alternative Approach to Recover Flipped Dependences ......... 87
4.3 Implementation .......................................................... 91
4.4 Evaluation ............................................................. 96
  4.4.1 Methodology ..................................................... 96
  4.4.2 Runtime Characteristics ....................................... 97
  4.4.3 Performance ..................................................... 101
4.5 Contributions and Impact .............................................. 103
5. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future ............................................. 105
   5.1 Problem and Motivation ............................................. 105
      5.1.1 More on memory models ..................................... 107
      5.1.2 Exposing Weak Memory Model Behaviors .................... 111
   5.2 Preliminaries: Common Notation ................................. 114
   5.3 Prior Work: Adversarial Memory .................................. 115
   5.4 Prescient Memory .................................................... 116
   5.5 Making Prescient Memory Practical ............................... 119
      5.5.1 Profiling Potential Future Values ............................ 120
      5.5.2 Predicting Future Values .................................... 121
      5.5.3 Fuzzy Deterministic Replay .................................. 124
   5.6 Implementation ..................................................... 125
   5.7 Evaluation .......................................................... 127
      5.7.1 Methodology .................................................. 127
      5.7.2 Stale and Future Values Found in Real Applications ........ 127
      5.7.3 Exposing Erroneous Behavior ................................ 129
      5.7.4 Run-Time Performance ....................................... 138
   5.8 Contributions and Impact .......................................... 139

6. Related Work ......................................................... 141

7. Conclusion .......................................................... 145
   7.1 Summary ........................................................... 145
   7.2 Impact and Meaning ................................................ 146

Bibliography .......................................................... 148
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>All possible state transitions for last-access states. T, T1, and T2 denotes different threads. *An upgrading transition to RdSh co gets the counter value c from a monotonically increasing global counter. A read by T of an object in the RdSh co state requires a fence transition if and only if a per-thread counter T.rdShCount &lt; c [25].</td>
</tr>
<tr>
<td>3.2</td>
<td>All possible state transitions for the hybrid state model. Instances of “OR” indicate cases in which a state can potentially transition between pessimistic and optimistic states. *Pessimistic uncontended transitions from RdSh c to RdShRLock(+) also update T.rdShCount to max(T.rdShCount, c).</td>
</tr>
<tr>
<td>3.3</td>
<td>The total number of spawned threads and maximum number of live threads for each program.</td>
</tr>
<tr>
<td>3.4</td>
<td>State transitions for hybrid tracking, compared with state transitions for optimistic tracking alone (shown in parentheses).</td>
</tr>
<tr>
<td>3.5</td>
<td>Run-time overhead of hybrid tracking with varying parameters for programs that showed statistically significant sensitivity. The default value and overhead are in bold.</td>
</tr>
<tr>
<td>4.1</td>
<td>Characteristics of recorded executions for Roctet and RegPlay. The columns named “Total” show the total numbers of recorded dependences for each analysis.</td>
</tr>
<tr>
<td>4.2</td>
<td>Average number of replay attempts until replay succeeds using the iterative patching approach. *These programs utilize extra techniques after profiling to reduce flipped read–write dependences.</td>
</tr>
<tr>
<td>5.1</td>
<td>Statistics of the number of distinct fields (non-reference type) that can load stale or future values for each program.</td>
</tr>
</tbody>
</table>
5.2 Summary of erroneous program behaviors discovered by returning stale or future values. *The program in Figure 5.5 is data race free, so AM and the PM workflow do not instrument any memory accesses. . . . . . . . . . . . . . 129
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Pseudocode for optimistic tracking’s instrumentation slow path (for program stores only) and coordination. T is the executing thread.</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Deferred unlocking encounters contention only for object-level data races. Comments show instrumentation actions assuming O is in pessimistic states.</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>High-level state transition diagram for the hybrid state model. The left and right halves show transitions starting in pessimistic and optimistic states, respectively. The diamonds on the vertical dashed line indicate decisions by the adaptive policy, described in Section 3.5.</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>The instrumentation fast path for hybrid tracking, for program stores only. (Handling loads is analogous but more complex.)</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>The instrumentation slow path for hybrid tracking for program stores.</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>Instrumentation at PSROs and responding safe points, for program stores only.</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>The challenge of recording pessimistic conflicting transitions.</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>Challenge of building an RS enforcer using hybrid tracking.</td>
<td>38</td>
</tr>
<tr>
<td>3.9</td>
<td>Cumulative distribution of conflicting transitions (explicit coordination only) triggered per object for optimistic tracking. Both axes use a logarithmic scale. The legend sorts programs by their maximum y-axis value. Three programs have a conflict rate &lt;0.0001% and are excluded.</td>
<td>48</td>
</tr>
<tr>
<td>3.10</td>
<td>Run-time overhead of pessimistic and optimistic tracking, compared with hybrid tracking. Each bar is the median of 20 trials. The intervals are 95% confidence intervals centered at the mean. Overheads exceeding 120% are labeled using two significant figures.</td>
<td>52</td>
</tr>
</tbody>
</table>
3.11 Run-time overhead of tracking alone on microbenchmarks. 54

3.12 Run-time overhead of optimistic and hybrid runtime support. 55

4.1 A transitively implied cross-thread dependence. The write–read dependence on x is transitively implied by the write–read dependence on y. 65

4.2 Overview of RegPlay’s record algorithm. (a) RegPlay records a dependence from end of a region to current access for a write–write or write–read dependence that does not constitute a region conflict. (b) RegPlay records the value of a read access if the read triggers a region conflict with last write. (c) For a write–write dependence involved in a region conflict, RegPlay resolves the region conflict via coordination. 69

4.3 Example of a flipped read–write dependence in replay and how RegPlay handles it. (a) During record, the read–write dependence from T2’s read to T1’s second write is not recorded. (b) During replay, if the read–write dependence is flipped, RegPlay would detect this inconsistency and return a correct value at the read. 70

4.4 An example transitive dependence involving three threads. The write–read dependence on x between T1 and T3 is transitively implied by the dependences between T1 and T2 on x, and between T2 and T3 on y. 84

4.5 A case that could cause a subtle deadlock after several replay attempts. The figure shows the interleaving during record. 91

4.6 Run-time overhead of record for RegPlay and Roctet on top of unmodified Jikes RVM. Thin vertical lines are 95% confidence intervals centered at the mean. 102

5.1 An assertion failure is possible under Java’s memory model. Existing dynamic analyses can expose the assertion failure. 106

5.2 An assertion failure is possible under Java’s memory model. Existing dynamic analyses cannot expose the failure. 106

5.3 An example program that can generate a divide-by-zero exception under HBMM. 107
5.4 Using stale values, the execution may not terminate. Using future values, the assertion can fail. .......................... 109

5.5 An example data-race-free program that can fail its assertion under HBMM but not DRF0 [3, 21, 22]. .......................... 109

5.6 An example out-of-thin-air result [22, 92, 128]. ......................... 110

5.7 An example program in which compiler transformations can violate JMM [128]. 111

5.8 An assertion failure is not possible under Java’s memory model. Dynamic analysis must be careful not to allow an execution in which the assertion fails. 112

5.9 Illustration (inspired by prior work [54]) of the behaviors permitted by various memory models (solid lines), exposed by dynamic analyses (dashed lines and bold text), and exposed by typical JVMs (dotted line). .......................... 113

5.10 Overview of the PM workflow. Dashed lines separate distinct program executions. .......................... 120

5.11 An example program for which the PM workflow can cause the assertion to fail. .......................... 124

5.12 Code from pjbb2005. .......................... 136

5.13 Run-time overhead of AM, PM-profiler, and PM. Each bar is the median of 10 trials. The intervals are 95% confidence intervals centered at the mean. Overheads exceeding 700% are labeled using two significant figures. .......................... 138
Chapter 1: Introduction

Software becomes increasingly parallel in order to scale with the successive microprocessor generations that provide more, instead of faster, processing cores. However, writing and debugging parallel programs is notoriously difficult. Parallel programs could experience various challenging concurrency issues, such as deadlock, data race, atomicity violation, and nondeterminism. General-purpose programming languages provide shared memory and locks, which are simple to understand, but hard to use to achieve both correctness and scalability.

Data races are among the most problematic concurrency bugs. A fundamental problem is that shared-memory languages and hardware memory models provide few, if any, guarantees for executions containing data races, in exchange for flexible support for high-performance implementations that require various compiler transformations and hardware optimizations. As a result, data races lead to ambiguous, unexpected, and erroneous behaviors in modern languages and hardware. For example, C/C++ executions that are racy (i.e., have a data race) have undefined semantics [3, 21]. Java provides defined but weak semantics for racy executions, in an effort to preserve memory and type safety [92], although later work has shown that this model is impractical to enforce [22, 92, 128]. In addition, data races can be the root cause of many other concurrency bugs, such as atomicity violation, order violation, and unexpected nondeterminism [87].
Data races and their erroneous effects occur nondeterministically and only under certain conditions. They are difficult to avoid, detect, fix, reproduce, and eliminate; programmers often introduce them intentionally for performance [19, 20, 72, 74]. Data races and their erroneous effects are thus ubiquitous, even in mature software systems [87].

Researchers have developed program analyses and software systems for concurrency correctness properties to help support reliable, scalable parallelism (this thesis uses the general term “runtime support” to refer to such dynamic analyses and software systems), but many of these tools have inherently limitations when they deal with executions containing data races. Notable examples of runtime support include data race detectors (e.g., [14, 16, 23, 31, 39, 40, 46–49, 53, 67, 73, 93, 107, 113, 121, 133, 138, 140, 147]), software transactional memory (e.g., [42, 61, 63, 130, 149]), systems that enforce strong memory models or tolerate data races (e.g., [94, 109, 131, 146]), atomicity checkers (e.g., [15, 50, 52, 55–57, 88, 91, 141]), analyses that expose concurrency bugs (e.g., [19, 29, 54, 72, 74, 103, 106, 124]), multithreaded record & replay (e.g., [10, 45, 58, 64, 65, 68, 79–81, 85, 98, 102, 110, 112, 118, 137, 142, 143]), and deterministic execution (e.g., [13, 43, 86, 108]). When dealing with racy executions, many existing analyses suffer from limitations that impede their widespread use. Static analyses typically suffer from low precision (i.e., they report many false positives) (e.g., [101]) and poor scalability (e.g., [100]). Dynamic analyses usually offer higher precision and better scalability than static analyses, but they commonly have limitations in terms of performance and coverage. First, many software-only dynamic analyses incur substantial performance overhead (e.g., [13, 42, 46, 53, 56, 80, 133]). Some approaches incur acceptable overhead, but they introduce other limitations such as ignoring data races (e.g., [58, 118]), or relying on non-existent custom hardware (e.g., [64, 65, 91, 94, 98, 131, 146, 146]). Second, dynamic analyses have limited coverage of program behaviors allowed in language and hardware.
memory models. Analyses that attempt to expose concurrency bugs typically only explore a small subset of all possible behaviors for racy executions (e.g., [29, 54, 74]).

1.1 Overview and Outline

This thesis targets the performance and coverage challenges faced by existing dynamic analyses for commodity systems. We explore several novel dynamic program analyses, in order to overcome or ameliorate the deficiencies of existing work in dealing with racy executions.

Chapter 2 first describes problems with data races and memory models, and their implications and challenges imposed on existing analyses and systems.

Existing runtime support needs to track (i.e., detect or control) an execution’s cross-thread dependences accurately, in order to soundly handle racy executions. Tracking cross-thread dependences typically slows programs substantially. Prior work tracks cross-thread dependences either “pessimistically,” slowing every program access, or “optimistically,” allowing for lightweight instrumentation of most accesses but dramatically slowing accesses involved in cross-thread dependences. Tracking cross-thread dependences poses a major performance limitation that impedes the widespread use of many runtime support.

Chapter 3 presents hybrid tracking, a framework for efficient tracking of cross-thread dependences. Hybrid tracking seeks to hybridize pessimistic and optimistic tracking in order to get the best of both world. A fundamental mismatch between pessimistic and optimistic tracking hinders their straightforward hybridization. Hybrid tracking addresses this challenge based on insights about how dependence tracking and program synchronization interact. An adaptive, profile-based policy makes run-time decisions about switching between pessimistic and optimistic tracking. We also build hybrid-tracking-based versions of a dependence
recorder and a region serializability enforcer, to demonstrate hybrid tracking is suitable for building efficient runtime support. Evaluation shows that hybrid tracking enables runtime support to overcome the performance limitations of both pessimistic and optimistic tracking alone. Hybrid tracking demonstrates an efficient and flexible approach for tracking cross-thread dependences, improving the performance for many runtime support that needs to correctly handle racy executions.

While hybrid tracking provides a general framework that efficiently tracks all dependences for any runtime support, a specific runtime support could further optimize its dependence tracking mechanism based on its need. In particular, hybrid tracking could be an overkill for an analysis that does not need to track all dependences.

Chapter 4 demonstrates how to optimize dependence tracking in the context of multithreaded record & replay. Nondeterminism in multithreaded programs significantly complicates software debugging and certain system designs. Researchers have proposed record & replay to overcome nondeterminisms. However, existing solutions suffer from various limitations, such as inability to provide replay determinism for racy executions, slowing programs by an order of magnitude, relying on unrealistic custom hardware. We introduce RegPlay, an efficient multithreaded record & replay algorithm that ensures deterministic replay for both data-race-free and racy executions. Our key insight is that not all cross-thread dependences are required to be recorded in order to achieve replay determinism. RegPlay employs lock-free instrumentation that only records necessary write–read and write–write dependences, avoiding the significant overhead to track read–write dependences. RegPlay ensures replay determinism by enforcing recorded dependences, while detecting and resolving violations of unrecorded read–write dependences. Evaluation shows that RegPlay records much fewer dependences and incurs low overhead comparing to a recent record &
replay approach, while preserves replay determinism. RegPlay advances the state of the art in multithreaded record & replay, overcoming limitations inherent to most approaches.

In addition to high performance overhead in tracking cross-thread dependences in a racy execution, programmers and existing analyses in fact have limited understanding of an execution with data races. A racy execution can exhibit a wide range of behaviors in a weak memory model. Researchers have introduced dynamic analyses that expose these behaviors, but these approaches fail to expose behaviors due to loading a “future value”—a value written by a program store that executes after the program load that uses the value.

Chapter 5 introduces prescient memory (PM), a dynamic analysis that exposes unexplored, potentially harmful behaviors of racy executions. PM is a novel dynamic analysis that exposes behaviors due to future values. PM speculatively returns a future value at a program load, and tries to validate the speculative value at a later store. To enable PM to expose behaviors due to future values in real application executions, we introduce a novel approach that increases the chances of using and successfully validating future values, by profiling and predicting future values and guiding execution. Experiments show that our approach is able to uncover a few previously unknown behaviors due to future values in benchmarked versions of real applications.

1.2 Contributions and Impact

This thesis demonstrates several novel approaches to overcome key limitations of existing work on concurrency correctness. We identify several key challenges for runtime support to soundly and efficiently handle executions containing data races, and provide solutions that advance the state of art. By providing approaches that leverage commodity hardware, coupled with favorable experimental results, this thesis shows promising directions to build
efficient, practical, scalable and cost-effective runtime support for concurrency correctness properties.

Hybrid tracking is the first approach that effectively and efficiently combines pessimistic and optimistic tracking for building runtime support. RegPlay is the first multithreaded record & replay algorithm that ensures replay determinism and supports both offline and online replay using only write–write and write–read dependences. Hybrid tracking and RegPlay both demonstrate better performance than existing approaches without compromising soundness or other guarantees. Prescient memory is the first dynamic analysis that exposes behaviors due to future values in large, real programs, broadening the range of program behaviors that software-only dynamic analyses are able to expose for racy executions. Prescient memory helps researchers and developers better understand real applications in weak memory models, aiding them to develop more stringent language and hardware specifications and reliable software.

In the coming years, most software systems will be parallel, and analyses and runtime support that improve software reliability, performance, and productivity of software development will have far-reaching impact. The proposed approaches promote the advancement and adoption of such analyses and runtime support, which will in turn benefit many areas of society that rely on dependable, high-performing parallel computing systems.
Chapter 2: Background

2.1 Data Race

In a shared-memory multithreaded program, a data race occurs when two threads access the same memory location without synchronization, and at least one of the accesses is a store [6]. In other words, the two accesses are conflicting (the same memory location accessed by two different threads, and at least one is a store), and not ordered by the happens-before relation, a partial order that is the union of thread and synchronization order [77].

An obvious effect for a data race is that the two accesses may execute in either order, leading to unpredictable, nondeterministic program state. For example, if the race consists of a load and a store, then the load may return either the old or the newly stored value. As a result, it may silently corrupt program data, produce confusing errors, or cause other errors such as atomicity or determinism violation. As we will describe further, a data race could bring about more suprising and counterintuitive results in relaxed memory models.

Detecting data races Researchers have introduced a plethora of approaches for detecting data races (e.g., [1, 14, 16, 23, 31, 39, 40, 46–49, 51, 53, 67, 73, 93, 100, 101, 107, 113, 114, 121, 133, 138, 140, 147]). Nevertheless, there exists a fundamental tradeoff between coverage
(detecting as many races as possible), precision (no false positives), and performance (overhead and scalability). All static analyses and some dynamic analyses have high coverage but report false positives, which developers find unacceptable [23,93]. On the other hand, dynamic analyses are typically precise but struggle to deliver acceptable performance.

This tradeoff is compounded by a second challenge: data races are sensitive to thread interleavings, program inputs, and execution environments, so they often manifest only in production runs, even for extensively tested programs [135]. A data race may require tens or hundreds of runs or more to manifest [150], and it can take weeks to reproduce, diagnose, and fix data races that occur in production systems [60,87].

**Data races are ubiquitous and harmful.** Data races are hard to avoid, detect, reproduce, and eliminate. They are widespread even in mature software [87]. Programmers often introduce data races intentionally in their efforts to improve performance and scalability and avoid deadlock, and these data races lead to a variety of unexpected, erroneous behaviors [19, 20, 29, 54, 72, 74, 87].

Even if all data races in a program are identified by data race detectors, fixing all of them still seems impractical. First, many large real-world applications commonly contain intentional data races on certain frequent accesses to avoid the cost of synchronization operations. The performance degradation of fixing these intentional data races is prohibitive in production settings [49,72,74,103]. Second, diagnosing and fixing data races involves significant amount of human labor (in days or even weeks [60]), so that time-constrained developers are less likely to invest much effort in this low-reward activity. Third, many data races, including the intentional data races, are assumed “benign” or “harmless” because they do not appear to affect program correctness or induce unexpected behaviors under
many executions. As we will explain, this is a common misconception because languages and hardware provide few guarantees for executions with data races. An assumed benign data race could become destructive in a different configuration, or in a future generation of compiler and hardware [19].

2.2 Memory Models

Sequential Consistency. An execution is \textit{sequentially consistent} (SC) if all memory accesses appear to be interleaved in an order that is consistent with each thread’s program order [78]. In an SC execution, each load must return the value of the most recent store to the same address by any thread.

A memory model defines the possible values for a load from shared memory [3]. A simple memory model is the \textit{sequentially consistent memory model} (SCMM), in which every execution must be SC. An SC execution is intuitive for programmers and analyses to reason about, since the execution is equivalent to interleave each thread’s loads and stores in a global sequential order. Nevertheless, it is expensive and impractical to enforce SCMM, which significantly restricts compiler and hardware optimizations that would require reordering memory accesses.

Researchers and engineers have defined \textit{relaxed memory models} [4] (also known as \textit{weak memory models}) for hardware and languages. A relaxed memory model permits a load to return any value among multiple legal values, which enables optimizations that need to reorder memory accesses. This thesis is concerned with \textit{language memory models} that must be enforced end-to-end, i.e. by the compiler and hardware with respect to the original source-level program. In contrast, hardware memory models only guarantee behaviors with respect to the compiled program.
DFR0. In 1990, Adve and Hill introduced the *data-race-free-0* (DRF0) memory model [5], which guarantees sequential consistency for well-synchronized program executions, i.e., executions that are *data-race-free* (DRF). The rationale for the DRF0 model is that it permits compilers and hardware to perform aggressive intra-thread optimizations, as long as they do not arbitrarily reorder memory accesses across synchronization operations. As long as programmers avoid data races, the effects of optimizations will not be externally visible. Modern shared-memory languages including Java and C++ provide memory models that are based on DRF0 [3, 21, 92]. In fact, most DRF0-based memory models provide serializability of *synchronization-free regions* (SFRs) for DRF executions — a stronger guarantee than SC [3, 5, 21, 92].

A fundamental problem for DRF0-based memory models is that they provide few, if any, guarantees for an execution containing data races. C and C++ lend undefined semantics to a “racy” execution (i.e., an execution with a data race) [21]. While this situation may be acceptable for unsafe languages such as C and C++, preserving memory and type safety in a safe language such as Java demands providing some semantics for racy executions.

The Java memory model (JMM) ensures certain weak semantics for racy program executions [92]. However, subsequent work shows that JMM actually precludes common Java virtual machine (JVM) optimizations [3, 22, 128]. Commercial JVMs thus *violate* JMM, since existing art does not demonstrate how to avoid certain undesirable results without seriously inhibiting optimizations [22, 128]. Recent work tries to address this issue, using techniques based on event structure models, but it is unclear whether this work perfectly delineates behaviors needed for optimization from other behaviors [70, 111]. These efforts show the difficulty of providing any semantics for racy executions without a dramatic departure from DRF0-based memory models.
Adve and Boehm [3] emphasize the significance of this problem:

*The inability to define reasonable semantics for programs with data races is not just a theoretical shortcoming, but a fundamental hole in the foundation of our languages and systems.*

### 2.3 Implication on Analyses and Runtime Support

Since data races are widespread in real software, practical dynamic program analyses and runtime support must handle executions containing data races. Runtime support that ignores data races is unsound and unreliable. For an analysis that helps developers debug programs, a data race might be the source of the problem that the developer is diagnosing. For an analysis that enforces a certain property, a data race could silently violate the property. For an analysis that exposes program errors, a data race could lead to a variety of unexpected behaviors. In all cases, ignoring data races would lead to incorrect results for the analyses. However, it is challenging for most runtime support to handle racy executions efficiently and soundly. Moreover, existing analyses have limited understanding about possible behaviors of data races.

**Performance.** Most existing runtime support for commodity systems (often called *software-only*) needs to track (detect or control) *cross-thread dependences* (data dependences involving two threads) soundly in order to achieve its functionality (e.g., checking for certain concurrency bugs, or enforcing certain concurrency correctness properties). For DRF executions, runtime support can track cross-thread dependences by instrumenting only program synchronization operations, because most DFR0-based memory models provide atomicity of SFRs. However, programs routinely have data races, so runtime support must instrument all potentially racy memory accesses. This instrumentation is particularly costly because it must add its own synchronization in order to ensure soundness in the presence
of data races in the program execution. Although sound static analysis can identify some accesses as definitely DRF, instrumenting the remaining potentially racy accesses is still expensive [36, 47, 80, 139]. Most existing runtime support uses an atomic operation at every access (e.g., [53, 56, 61, 79, 80]). Such frequent synchronization typically slows executions by several times or more.

Coverage. The ubiquitous existence of data races and the difficulty in eliminating all of them, have motivated researchers to develop and evaluate analyses that expose behaviors of racy executions, so that developers can triage and classify data races based on their harmfulness [29, 54, 72, 74, 103, 124]. These analyses have demonstrated that many real data races could lead to unexpected harmful behaviors in weak memory models.

The conventional wisdom for decades, persisting even to the present day, is that many data races are “benign.” This misconception has been solidified in part by prior work that exposes behaviors in racy executions, but considers only SC behaviors [72, 103, 124]. Prior work shows that real, seemingly benign data races can lead to harmful behaviors under certain compilers [19]. A well-behaved program with a data race can become incorrect following a change in compiler or environment. More recent research considers non-SC behaviors, and shows that many data races are demonstrably harmful [29, 54, 74]. However, existing analyses have not exposed the full range of possible behaviors allowed in weak memory models, so they could still fail to expose certain potentially harmful effects due to data races.

To sum up, existing analyses and systems for concurrency correctness are far from optimal for achieving both performance and soundness. In addition, the full extent of possible behaviors for racy executions in real programs remains yet to be explored. Practical solutions
and advancement that work in existing weak memory models would not only immediately address these challenges and improve today’s software, but also help the study and adoption of strong memory models that might fundamentally solve the problems due to data races in the future.
Chapter 3: Hybrid Tracking: Combining Pessimistic and Optimistic
Tracking of Cross-Thread Dependences

This chapter introduces a general approach called hybrid tracking that addresses the performance problem faced by various existing runtime support (Section 2.3). Hybrid tracking combines two existing dependence tracking mechanisms, pessimistic and optimistic tracking, in order to get the benefits of both.

3.1 Problem and Motivation

Tracking cross-thread dependences. As explained in Section 2.3, runtime support for concurrency correctness must track cross-thread dependences, which are data dependences (write–read, write–write, and read–write dependences) involving two threads, in order to soundly deal with executions containing data races. Tracking dependences means doing one of the following soundly (i.e., without missing dependences):

• Detect (monitor) dependences. Examples: data race detectors, atomicity violation detectors, and dependence recorders (e.g., for record & replay).

• Control (enforce) dependences. Examples: transactional memory, enforcing memory models, and deterministic execution.
To track cross-thread dependences, instrumentation at each memory access maintains the last-access state of the accessed object. (This chapter uses the term “object” to refer to any unit of shared memory.) Without loss of generality, we assume dependence tracking uses the following per-object states:

- **WrEx_T**: Write exclusive for thread T. Last read or written by T.
- **RdEx_T**: Read exclusive for T. Last read (not written) by T.
- **RdSh_c**: Read shared. Last read by multiple threads. The value c helps ensure sound tracking of write–read dependences. Prior work that introduces the counter provides details on how it helps enable sound tracking of cross-thread dependences [25].

Table 3.1 shows all possible state transitions, each of which is triggered by a program read or write by some thread. Prior work shows that these state transitions establish happens-before edges [77] that transitively imply all of an execution’s cross-thread dependences [25].

*Same-state* transitions involve no state change; they do not imply any cross-thread dependences. Other transitions imply potential cross-thread dependences. *Upgrading* transitions change the state to a new state that permits accesses allowed under the old state. *Fence* transitions enable detecting write–read dependences when a thread reads a RdSh_c object for the first time (prior work provides details [25], which are not integral to understanding this chapter). Finally, *conflicting* transitions change the old state to a new state that disallows accesses allowed under the old state.

**Instrumentation atomicity.** To track dependences accurately, instrumentation at each memory access must check, and potentially update, the accessed object’s state. These actions must appear to happen together *atomically* to avoid missing dependences; we
<table>
<thead>
<tr>
<th>Transition type</th>
<th>Old state</th>
<th>Access</th>
<th>New state</th>
<th>Sync. required</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pessimistic tracking</td>
</tr>
<tr>
<td>Same state</td>
<td>WrEx_{T}</td>
<td>R/W by T</td>
<td>Same</td>
<td>CAS</td>
</tr>
<tr>
<td></td>
<td>RdEx_{T}</td>
<td>R by T</td>
<td>Same</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh_{c}</td>
<td>R by T</td>
<td>Same*</td>
<td></td>
</tr>
<tr>
<td>Upgrading</td>
<td>RdEx_{T}</td>
<td>W by T</td>
<td>WrEx_{T}</td>
<td>CAS</td>
</tr>
<tr>
<td></td>
<td>RdEx_{T1}</td>
<td>R by T2</td>
<td>RdSh_{c}</td>
<td></td>
</tr>
<tr>
<td>Fence</td>
<td>RdSh_{c}</td>
<td>R by T</td>
<td>Same*</td>
<td>CAS</td>
</tr>
<tr>
<td>Conflicting</td>
<td>WrEx_{T1}</td>
<td>W by T2</td>
<td>WrEx_{T2}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WrEx_{T1}</td>
<td>R by T2</td>
<td>RdEx_{T2}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdEx_{T1}</td>
<td>W by T2</td>
<td>WrEx_{T2}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh_{c}</td>
<td>W by T</td>
<td>WrEx_{T}</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: All possible state transitions for last-access states. T, T1, and T2 denotes different threads. *An upgrading transition to RdSh_{c} gets the counter value c from a monotonically increasing global counter. A read by T of an object in the RdSh_{c} state requires a fence transition if and only if a per-thread counter T.rdShCount < c [25].

call this property *instrumentation atomicity*. Furthermore, most runtime support requires *instrumentation–access atomicity*: that the instrumentation and access appear to execute together atomically. (A notable exception is data race detection, which requires only instrumentation atomicity because it does not need to know the order of racy accesses.)

In any case, instrumentation atomicity and instrumentation–access atomicity incur similar costs.

To guarantee instrumentation–access atomicity, most existing runtime support uses instrumentation that performs atomic operations at every memory access, which we call *pessimistic tracking* (Section 3.1.1). Alternatively, *optimistic tracking* eschews atomic operations at non-communicating accesses, but requires inter-thread coordination at some communicating accesses (Section 3.1.2).
We emphasize that the instrumentation and per-object states used by dependence tracking, as well as the synchronization needed to ensure instrumentation–access atomicity, are visible to runtime support only, not to programmers.

3.1.1 Pessimistic Tracking

Pessimistic tracking provides instrumentation–access atomicity via a small critical section around each access and its instrumentation. As Table 3.1 indicates, pessimistic tracking requires an atomic operation (e.g., compare-and-swap instruction) at every access. The following pseudocode shows typical instrumentation at a program store. (Instrumentation at a load is similar but more complex since there are more possible state transitions.)

```c
do {
    s = o.state; // load per-object metadata
} while (s == LOCKED || !CAS(&o.state, s, LOCKED));
if (s != WrEx_T) { // T is the executing thread
    /* handle potential cross-thread dependence(s) */
}
o.f = ...; // program store
memfence; // type of fence depends on program access type
o.state = WrEx_T; // unlock and update metadata
```

The instrumentation starts a critical section by “locking” the object’s state (represented as `o.state`) using a special LOCKED value. If the current state is any state other than `WrEx_T` (T is the current executing thread), a potential cross-thread dependence exists, requiring additional runtime-support-specific work (not shown). For example, a dependence recorder could record the dependence in a log, and speculation-based enforcement of region serializability could roll back and restart a code region.

The atomic operation `CAS(addr, oldVal, newVal)` attempts to update `addr` from `oldVal` to `newVal`, returning true on success.
**Performance.** Pessimistic tracking requires frequent atomic operations and memory fences, which slow program execution substantially by triggering remote cache misses and serializing out-of-order execution. In our experiments on benchmarked versions of large, real-world Java programs, pessimistic tracking (without any runtime support on top of it) slows programs by more than 4X (>300% overhead) on average (Section 3.6.5).

Existing runtime support commonly employs pessimistic tracking (e.g., [53, 56, 61, 79, 80]). We note that existing approaches often avoid performing an atomic operation for every memory access. For example, software transactional memory (STM) [61] can use instrumentation that avoids atomic operations for accesses to the same object in the same transaction. Some STM systems can further avoid atomic operations at loads by validating them lazily, but still require memory fences (e.g., [120]). Data race detectors [53] can avoid atomic operations for repeated accesses in the same synchronization-free region. Nonetheless, atomic operations and memory fences remain frequent enough to incur high overhead. Other approaches have sidestepped explicit dependence tracking but incur other limitations and costs, e.g., DoublePlay detects conflicts implicitly using speculation and replication, but it adds high overhead unless extra cores are available [137].

### 3.1.2 Optimistic Tracking

In contrast, optimistic tracking avoids synchronization at most accesses. Prior work uses optimistic tracking either to implement program locks [30, 75, 119] or to track cross-thread dependences [25, 122, 138]. This project focuses on the latter context.

Optimistic tracking provides instrumentation–access atomicity without requiring synchronization at accesses that trigger no state change, but it requires coordination at accesses that trigger conflicting state changes. Table 3.1 shows the differing kinds of synchronization
needed for each transition type. The following pseudocode shows the instrumentation added at a program store (instrumentation for a load is similar but more complex):

```java
if (o.state != WrEx_T) {
    // fast path
    slowPath(o);
} 

if f = ...; // program store
```

If the object’s state is already \( \text{WrEx}_T \), the instrumentation takes the synchronization-free fast path. Otherwise, the instrumentation executes the slow path, shown in Figure 3.1, which changes the state and handles the possible cross-thread dependence. Upgrading transitions require an atomic operation to avoid racing with other threads changing the state. Fence transitions require a memory fence to ensure visibility for write–read dependences.

**Conflicting transitions require coordination.** For conflicting transitions (last four rows of Table 3.1), it is insufficient to change the state using an atomic operation. Suppose a thread \( T \) wants to write to an object in \( \text{RdEx}_{\text{remote}T} \) or \( \text{WrEx}_{\text{remote}T} \) state. If \( T \) changes the state to \( \text{WrEx}_T \)—even if it uses an atomic operation—and accesses the object, it may conflict with \( \text{remote}T \)’s continued unsynchronized instrumentation and access to the same object, violating instrumentation–access atomicity. Instead, a thread triggering a conflicting transition must coordinate with thread(s) that can access the object under the old state.

Figure 3.1 shows the instrumentation slow path, for a program store only. To initiate coordination, the executing thread \( T \) first changes the object’s state to an intermediate state \( \text{Int}_T \) (line 8), which simplifies the protocol by allowing only one thread at a time to initiate coordination for an object. \( T \) then coordinates with the remote thread (line 12) to ensure that \( T \)’s state change does not interrupt the remote thread’s instrumentation–access atomicity. If \( o.\text{state} \) is \( \text{RdSh}_c \), \( T \) conservatively coordinates with every other thread.
slowPath(o) {
  state = o.state;
  if (state == RdExT) {
    ...; // upgrading transition to WrExT
    return;
  }
  // Coordination for conflicting transition:
  while (state == Int∗ || !CAS(&o.state, state, IntT)) {
    checkAndRespondToRequests(); // non-blocking safe point
    state = o.state; // re-read state
  }
  coordinate(getOwner(state));
  o.state = WrExT;
}

coordinate(remoteT) {
  response = sendRequest(remoteT); // return true if implicit coordination used
  while (!response) {
    checkAndRespondToRequests(); // non-blocking safe point
    response = checkResponse(remoteT);
  }
}

Figure 3.1: Pseudocode for optimistic tracking’s instrumentation slow path (for program stores only) and coordination. T is the executing thread.

The remote thread, which we call remoteT, participates in coordination only when it is at a safe point: a program point that does not interrupt instrumentation–access atomicity (not shown in the figure). Conveniently, managed language VMs already place safe points at periodic points in compiled code (e.g., method entries and loop back edges) so threads can be stopped promptly, e.g., for stop-the-world garbage collection. Blocking operations, such as waiting to acquire a lock or for I/O, are also safe points. If remoteT is at a blocking safe point, T coordinates with remoteT implicitly by updating remoteT’s status atomically, which remoteT will see when it finishes blocking. Otherwise, T coordinates
with \texttt{remoteT} \textit{explicitly}: \texttt{T} sends a request to \texttt{remoteT}, and \texttt{remoteT} responds at its next safe point. (Figure 3.1 does \textit{not} show the actions of \texttt{remoteT}.) Whenever a safe point responds (implicitly or explicitly) to coordination request(s), it is called a \textit{responding safe point}. An important detail is that while \texttt{T} waits for an explicit coordination response, it acts as a safe point (line 18), so other threads can perform coordination with \texttt{T} in order to gain access to other objects, thus avoiding deadlock.

Finally, \texttt{T} changes the state to \texttt{WrEx}\textsubscript{T} (line 13) and proceeds with its access. Since \texttt{remoteT} coordinates only at a safe point, and \texttt{T} does not proceed with its access until coordination completes, instrumentation–access atomicity is preserved.

\textbf{Performance.} Optimistic tracking exploits a tradeoff: it avoids synchronization in the common, non-conflicting case but requires coordination in the uncommon, conflicting case. As Section 3.6.5 shows, for programs that perform little communication, optimistic tracking incurs low overhead. For programs that perform more communication (e.g., as little as 0.5\% of accesses conflicting), optimistic tracking incurs high overhead (e.g., >100\% run-time overhead). Optimistic tracking’s key limitation—and the main impediment to its widespread use—is its poor performance for all but low-conflict executions.

The following table reports costs of different kinds of state transitions, averaged across all programs (Section 3.6.2 describes overall experimental methodology):

<table>
<thead>
<tr>
<th></th>
<th>Pessimistic</th>
<th>Optimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same state</td>
<td>150</td>
<td>47</td>
</tr>
<tr>
<td>Conflicting</td>
<td></td>
<td>9,200</td>
</tr>
<tr>
<td>Explicit</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>Implicit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The average time in CPU cycles for pessimistic instrumentation is 150 cycles, which is largely independent of the transition type. Optimistic instrumentation’s cost is only a few dozen cycles for non-communicating accesses (\textit{Same state}), but conflicting transitions that
use *Explicit* coordination cost 2–3 orders of magnitude more by incurring the latency of roundtrip communication. *Implicit* coordination requires atomic operations but incurs no latency, so its cost is relatively close to the cost of a pessimistic access.

**Goal and outline.** Our goal is to develop a hybrid of pessimistic and optimistic tracking that keeps overhead low by using optimistic tracking for most accesses, but avoids most coordination by using pessimistic tracking for most conflicting accesses.

Sections 3.2 presents challenges inherent in combining pessimistic and optimistic tracking, and introduces a hybrid state model that addresses these challenges. Sections 3.3 and 3.4 design sound and efficient runtime support using the hybrid state model. Section 3.5 describes a policy that decides between pessimistic and optimistic states at run time. The remaining sections describe our implementation and evaluation.

### 3.2 Hybrid State Model

This section introduces a hybrid state model that combines the state models of pessimistic and optimistic tracking. Section 3.2.1 argues that hybridization presents fundamental challenges, and then describes insights for addressing these challenges. Section 3.2.2 presents details of the hybrid state model.

#### 3.2.1 The Pessimistic–Optimistic Mismatch

Pessimistic and optimistic tracking are fundamentally different in two key ways that complicate hybridization. First, pessimistic and optimistic tracking differ in how they transfer access privileges. Pessimistic tracking unlocks an object’s state after a program access, allowing another thread to lock the state. Optimistic tracking, on the other hand, does *not* unlock the state after an access; instead, a thread relinquishes access privileges only when
requested by another thread. To support objects being in both pessimistic and optimistic states, it seems that each access must be followed by potentially costly instrumentation that conditionally unlocks the state (depending on whether the state is pessimistic).

Second, pessimistic and optimistic tracking provide instrumentation–access atomicity differently. Pessimistic tracking provides atomicity of each instrumentation–access pair. Optimistic tracking provides atomicity interrupted at responding safe points—including conflicting accesses that respond to coordination requests. This mismatch implies that the atomicity of instrumented code can be interrupted at points that are statically unpredictable, making it problematic to design efficient runtime support that detects and controls cross-thread dependences. This problem is easier to understand in the context of specific kinds of runtime support; Sections 3.3 and 3.4 explain these challenges in the contexts of the dependence recorder and region serializability (RS) enforcer.

In the early stages of this work, we designed and implemented a straightforward approach for combining pessimistic and optimistic tracking. This approach added conditional instrumentation after every program access, to unlock the state when it was pessimistic. We built a dependence recorder and RS enforcer on top of this hybrid approach, but they added significant overhead to perform conditional instrumentation and to deal with atomicity being interrupted unpredictably at many program points. Overall, this straightforward hybrid approach was unable to outperform optimistic tracking on average.

To overcome the mismatch between pessimistic and optimistic tracking that impaired our initial design, we introduce the following insight: the hybrid state model can and should defer unlocking of pessimistic states. Deferring unlocking consists of the following design points:
• A thread defers unlocking pessimistic states until the next *program synchronization release operations* (PSRO) such as lock release, monitor wait, or thread fork.

• To avoid substantial false contention from concurrent readers, pessimistic states use *reader–writer locking*.

• A thread encountering any remaining contention “falls back” to using *coordination* to change an object’s state.

Interestingly, if instrumentation encounters contention trying to lock a pessimistic state, the access must be involved in an *object-level data race*: two unsynchronized, conflicting accesses to the same object, but not necessarily the same field or array element. An object-level data race is a necessary but insufficient condition for a true (precise) data race. Prior work shows that object-level data races closely over-approximate precise data races in practice [138]. The performance of our hybrid design relies on object-level data races being rare [23, 93, 138], so that few (if any) pessimistic transitions encounter contention.

Deferring unlocking bridges the pessimistic–optimistic mismatch by making pessimistic tracking more “optimistic”: threads do not unlock pessimistic states until PSROs, but incur high coordination cost (the same as for optimistic states) if a conflicting access occurs in the meantime.

**Example.** Figure 3.2 illustrates deferring unlocking of pessimistic states. The example assumes o is in pessimistic states for the accesses shown. In Figure 3.2(a), each thread executes a critical section acquiring the same program lock m. Code comments (e.g., /* lock o.state */) summarize the run-time behavior of hybrid tracking’s instrumentation. Immediately before T1 releases m (a PSRO), instrumentation unlocks all pessimistic states that T1 has locked, including o’s state. T2 thus locks o’s state without contention.
In contrast, in Figure 3.2(b), the two accesses are involved in an object-level data race (in this case, a true data race). As a result, T2 encounters contention when trying to lock o’s state. T2 handles this case safely by falling back to using coordination: T2 sends a coordination request to T1, which unlocks all pessimistic states at the next responding safe point, enabling T2 to lock o’s state.

3.2.2 States, Terminology, and Transitions

The hybrid state model uses the following states:
• Pessimistic states can be either unlocked or locked. The **pessimistic unlocked** states are $\text{WrEx}_{T}^{\text{Pess}}$, $\text{RdEx}_{T}^{\text{Pess}}$, and $\text{RdSh}_{c}^{\text{Pess}}$. The **pessimistic locked** states are $\text{WrEx}_{T}^{\text{RLock}}$, $\text{WrEx}_{T}^{\text{WLock}}$, $\text{RdEx}_{T}^{\text{RLock}}$, and $\text{RdSh}_{c}^{\text{RLock}(n)}$. To support reader–writer locking, a $\text{WrEx}_{T}$ state can be either read- or write-locked, and a $\text{RdSh}_{c}^{\text{RLock}(n)}$ state is read-locked by $n$ threads. The read-locked write-exclusive state ($\text{WrEx}_{T}^{\text{RLock}}$) enables a second concurrent reader to upgrade to $\text{RdSh}_{c}^{\text{RLock}(2)}$, instead of encountering contention. To support reentrant read locks, each thread also keeps track of the set of objects whose states it has read-locked.

• The optimistic states are $\text{WrEx}_{T}^{\text{Opt}}$, $\text{RdEx}_{T}^{\text{Opt}}$, and $\text{RdSh}_{c}^{\text{Opt}}$.

A **pessimistic (or optimistic) object** is an object whose state is pessimistic (optimistic). A **pessimistic (optimistic) access** is a program access to a pessimistic (optimistic) object. A **pessimistic (optimistic) transition** is a transition from a pessimistic (optimistic) state to another pessimistic (optimistic) state. The model also supports transitions between pessimistic and optimistic states.

Figure 3.3 shows at a high level the state transitions in the hybrid state model. The labeled circles summarize the three types of states: pessimistic unlocked, pessimistic locked, and optimistic. Arrows represent transitions between states: bold, red arrows show transitions requiring coordination; other transitions do not require coordination. The rest of this section further explains Figure 3.3, focusing on transitions that are different from those shown in Table 3.1. Section 3.2.3 shows pseudocode for hybrid tracking’s instrumentation. Section 3.2.4 presents a table detailing every state transition.

**Pessimistic uncontended transitions.** Any access to an unlocked pessimistic state triggers an **uncontended** transition to a corresponding locked state (see the transition labeled
Figure 3.3: High-level state transition diagram for the hybrid state model. The left and right halves show transitions starting in pessimistic and optimistic states, respectively. The diamonds on the vertical dashed line indicate decisions by the adaptive policy, described in Section 3.5.

“Any access (uncontended)” in Figure 3.3). For example, a read (or write) by T1 to an object in $\text{WrEx}_{T_1}^{\text{Pess}}$ state triggers an uncontended transition to $\text{WrEx}_{T_1}^{\text{RLock}}$ ($\text{WrEx}_{T_1}^{\text{WLock}}$). A read by T2 to an object in $\text{WrEx}_{T_1}^{\text{Pess}}$ triggers an uncontended transition to $\text{RdEx}_{T_2}^{\text{RLock}}$.

An access to a locked state that does not conflict with the state also triggers an uncontended transition (transition labeled “Non-conflicting access (uncontended & possibly reentrant)”). For example, a read by T2 to a $\text{RdEx}_{T_1}^{\text{RLock}}$ object triggers an uncontended transition to $\text{RdSh}_{T_2}^{\text{RLock(2)}}$ (read-locked by T1 and T2). A write by T1 to a $\text{WrEx}_{T_1}^{\text{RLock}}$ object triggers an uncontended transition to $\text{WrEx}_{T_1}^{\text{WLock}}$. If an uncontended transition requires no
state change at all (e.g., a read by \(T_1\) to an object in \(\text{RdEx}_{T_1}^{\text{RLock}}\) state), we also call the transition *reentrant*. Reentrant transitions require no atomic operations.

**Unlocking of pessimistic states.** To support deferred unlocking, each thread records every pessimistic object whose state it has locked in the thread’s *lock buffer*. Every program synchronization release operation (PSRO) and responding safe point *flushes* the buffer by unlocking the states of all objects in the buffer (transition labeled “PSRO & responding safe point”). Unlocking a \(\text{RdSh}_c^{\text{RLock}(n)}\) object means transitioning to \(\text{RdSh}_c^{\text{RLock}(n-1)}\) (if \(n > 1\)) or the unlocked state \(\text{RdSh}_c^{\text{Pess}}\) (if \(n = 1\)). Whenever a thread flushes its lock buffer, it also clears its set of read-locked objects.

**Pessimistic contended transitions.** An access that *conflicts* with a pessimistic locked state cannot immediately change the state. It triggers a *contended* state transition, which initiates coordination with the thread(s) that have locked the object’s state (transition labeled “Conflicting access (contended)”).

Since every responding safe point flushes the lock buffer, the thread(s) that have locked the state will unlock it, allowing the accessing thread to change the state into a compatible pessimistic locked state. By using coordination to trigger early unlocking of states, contended transitions ensure responsiveness and deadlock freedom when an execution violates deferred unlocking’s assumption of object-level data race freedom.

As an example, in Figure 3.2(b), a read by \(T_2\) to an object in \(\text{WrEx}_{T_1}^{W\text{Lock}}\) triggers a contended transition: \(T_1\) unlocks the state to \(\text{WrEx}_{T_1}^{\text{Pess}}\) before responding to coordination. \(T_2\) then performs an uncontended transition from \(\text{WrEx}_{T_1}^{\text{Pess}}\) to \(\text{RdEx}_{T_2}^{\text{RLock}}\).
if (o.state != WrExOpt) {
    slowPath(o);
}

o.f = ...; // program store

Figure 3.4: The instrumentation fast path for hybrid tracking, for program stores only. (Handling loads is analogous but more complex.)

Transitions between pessimistic and optimistic states. The model supports transitioning to an optimistic state whenever it unlocks a pessimistic state (upper diamond in Figure 3.3), and to a pessimistic state from an optimistic state on any conflicting transition (lower diamond).

Although we have designed and presented hybrid tracking based on the states and transitions in Table 3.1, our hybridization approach could in theory be applied to other optimistic and pessimistic approaches that use different state models to track dependences.

3.2.3 Instrumentation Pseudocode

Figures 3.4 to 3.6 show the instrumentation added by hybrid tracking. For simplicity, we only show instrumentation for a program store. The instrumentation for loads is more complex because it handles RdExT and RdShc states and supporting reentrant reader locks.

The fast path (Figure 3.4) only checks for the WrExOpt state, since we expect that the majority of accesses trigger same-state optimistic transitions. The slow path (Figure 3.5) changes the state based on hybrid tracking’s state transitions (Figure 3.3 and Table 3.2). The slow path repeatedly reloads and tries to change the state if an atomic update fails. A contended transition triggers coordination (line 39); then the slow path retries until the state becomes unlocked, enabling an uncontended transition (lines 33–37). Upon a successful
slowPath(o) {
    while(true) {
        state = o.state;
        if (isPess(state)) { // Pessimistic
            // Pessimistic Locked, uncontended
            if (state == WrEx\textsubscript{T}^{WLock}) break;
            // Pessimistic Unlocked
            if (isUnlocked(state) || state == WrEx\textsubscript{T}^{RLock}) {
                if (CAS(&o.state, state, WrEx\textsubscript{T}^{WLock})) {
                    T.lockBuffer.add(o);
                    break;
                }
            } else {
                // Pessimistic Locked, contended
                coordinate(getOwner(state));
            }
        } else { // Optimistic
            if (state == RdEx\textsubscript{T}^{Opt}) { ... }
            if ((state != Int\textsubscript{T}) && CAS(&o.state, state, Int\textsubscript{T})) {
                coordinate(getOwner(state));
                // Decision from adaptive policy
                if (AdaptivePolicy.toPess(o)) {
                    o.state = WrEx\textsubscript{T}^{WLock};
                    T.lockBuffer.add(o);
                } else {
                    o.state = WrEx\textsubscript{T}^{Opt};
                }
                break;
            }
        }
    }
    checkAndRespondToRequests(); // non-blocking safe point
}

Figure 3.5: The instrumentation slow path for hybrid tracking for program stores.
Figure 3.6: Instrumentation at PSROs and responding safe points, for program stores only.

transition to a pessimistic state, the instrumentation adds the object to the per-thread lock buffer (lines 35 and 48).

Figure 3.6 shows the instrumentation at each PSRO and responding safe point. The instrumentation flushes the current thread’s lock buffer by unlocking each object in the buffer, potentially transferring the object to an optimistic state, according to the adaptive policy (Section 3.5). The pseudocode shows how to handle objects in \( \text{WrEx}^\text{Opt}_T \) state only, not other states.

### 3.2.4 Complete State Transitions

Table 3.2 shows all possible transitions for the hybrid state model. Rows above the double line are pessimistic transitions; rows below the double line are optimistic transitions. The table essentially expands on Figure 3.3 to show the detailed transitions between specific pessimistic and optimistic states.

The first 16 rows are *Pessimistic uncontended* transitions that lock a pessimistic state before a program access. The old state is either the same as the new state (a *reentrant* transition), unlocked, or locked in a state that does not conflict with the access.

\(^2\) An early version of our work introduces a significantly different hybrid state model (e.g., it does not use deferred unlocking) and thus presents significantly different state transitions [33].
For *Pessimistic contended* transitions, a thread T2’s access will conflict with the object’s current locked state. T2 initiates coordination, so thread(s) that have locked the state will unlock it at their next responding safe point.

The *Pessimistic unlock OR Pess → Opt* rows show the transitions for deferred unlocking. These transitions occur at program synchronization release operations (PSROs) and responding safe points, instead of at program memory accesses. Unlocking a pessimistic state also provides an opportunity to transition an object to an optimistic state, according to the adaptive policy (Section 3.5).

The table omits some details for pessimistic read accesses. Each thread keeps track of which objects it has read-locked in a per-thread read set, T.rdSet. (1) When T reads an object not in its read set (o ∉ T.rdSet), it adds the object to its read set (T.rdSet ← T.rdSet ∪ {o}). (2) Whenever T flushes its lock buffer, it also clears its read set (T.rdSet ← ∅).

Optimistic state transitions are essentially the same as for optimistic tracking (Section 3.1.2). The *Conflicting* transition rows show that a conflicting access by T2 triggers the coordination protocol, which initially changes an object’s state to an intermediate state IntT. After coordination between threads, the hybrid model supports transitioning to a pessimistic or optimistic tracking state, based on the adaptive policy (Section 3.5).

### 3.3 Recording and Replaying Dependences

This section demonstrates how runtime support that needs to detect (i.e., monitor) cross-thread dependences soundly can use our hybrid state model. We build a *dependence recorder* based on hybrid tracking that identifies and records happens-before edges that transitively imply all cross-thread dependences in the execution.
<table>
<thead>
<tr>
<th>Trans. type</th>
<th>Old state</th>
<th>Program access</th>
<th>New state</th>
<th>Sync. needed</th>
<th>Cross-thread dependence?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pessimistic</td>
<td>WrEx**</td>
<td>R or W by T</td>
<td>Same</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>unconditioned</td>
<td>WrEx</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(reentrant)</td>
<td>RdEx</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>R by T if a ∈ T.TrSet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pessimistic</td>
<td>WrEx**</td>
<td>W by T</td>
<td>CAS</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>unconditioned</td>
<td>WrEx</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdEx</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>R by T if a ∉ T.TrSet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WrEx**</td>
<td>W by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WrEx</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdEx</td>
<td>W by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>W by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>W by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pessimistic</td>
<td>WrEx**</td>
<td>W by T</td>
<td>Handled at owner thread(s)' responding safe points</td>
<td>Roundtrip coordination</td>
<td>Maybe</td>
</tr>
<tr>
<td>contended</td>
<td>WrEx</td>
<td>W by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdEx</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>W by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pessimistic</td>
<td>WrEx**</td>
<td>PSRO or</td>
<td>CAS</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>unlock</td>
<td>WrEx**</td>
<td>responding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>RdEx*</td>
<td>safe point</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pess → Opt</td>
<td>RdSh*</td>
<td>if n &gt; 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>RdSh**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Same state</td>
<td>WrEx**</td>
<td>R or W by T</td>
<td>Same</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>RdEx*</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>R by T if T.rdShCount ≥ c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upgrading</td>
<td>WrEx**</td>
<td>W by T</td>
<td>CAS</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdEx*</td>
<td>R by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>RdSh**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fence</td>
<td>WrEx**</td>
<td>R by T if T.rdShCount &lt; c</td>
<td>(T.rdShCount ← c)</td>
<td>Memory fence</td>
<td>Maybe</td>
</tr>
<tr>
<td>Conflicting OR</td>
<td>WrEx**</td>
<td>W by T</td>
<td>ImT2 → WrExT2 OR WrEx** T2 Lock</td>
<td>Roundtrip coordination</td>
<td>Maybe</td>
</tr>
<tr>
<td>Opt → Pess</td>
<td>RdEx*</td>
<td>W by T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RdSh*</td>
<td>W by T</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: All possible state transitions for the hybrid state model. Instances of “OR” indicate cases in which a state can potentially transition between pessimistic and optimistic states. *Pessimistic uncontended transitions from RdSh\* to RdSh\*\* also update T.rdShCount to max(T.rdShCount, c).
3.3.1 Optimistic Dependence Recorder and Replayer

Multithreaded record & replay helps programmers debug nondeterministic multithreaded programs, and it provides systems benefits such as replication-based fault tolerance [79–81, 81, 110, 118, 137, 142]. Recording a multithreaded execution is expensive: detecting dependences requires synchronized instrumentation at each potentially racy access [79, 80]. Some approaches sidestep the problem of capturing cross-thread dependences explicitly, but introduce other limitations such as not supporting online or offline replay [81, 110, 142], or needing extra cores for speculation [137].

Prior work introduces a record & replay approach that designs (1) an optimistic recorder on top of optimistic tracking and (2) an optimistic replayer for the recorder [24, 25]. (The optimistic replayer is “optimistic” because it replays dependences recorded by the optimistic recorder. It does not use optimistic tracking.) The optimistic recorder identifies and records happens-before edges at transitions between $\text{WrEx}^{\text{Opt}}$, $\text{RdEx}^{\text{Opt}}$, and $\text{RdSh}^{\text{Opt}}$ states. It records each happens-before edge by recording its source and sink in per-thread logs. In another execution, the optimistic replayer replays each happens-before edge by making the sink wait for its corresponding source to be reached.

3.3.2 Hybrid Dependence Recorder & Replayer

We design a hybrid recorder based on hybrid tracking, and a hybrid replayer for the hybrid recorder. For optimistic transitions, the hybrid recorder uses the same approach as the optimistic recorder. For some, but not all, pessimistic transitions, the hybrid recorder uses essentially the same approach as for optimistic transitions, since pessimistic and optimistic states and transitions each maintain the same last-access information. For example, the
recorder can record a happens-before edge for $RdEx_{T_1}^{Pess} \rightarrow RdSh_{c}^{Rlock(2)}$ in the same way that it records $RdEx_{T_1}^{Opt} \rightarrow RdSh_{c}^{Opt}$.

**Pessimistic conflicting transitions.** The key challenge is pessimistic transitions that involve conflicting states, as Figure 3.7(a) shows. In this example, suppose pessimistic transitions do not defer unlocking. Thread $T_1$ immediately unlocks an object $o$ to $WrEx_{T_1}^{Pess}$ state after a write to $o$; then $T_2$ wants to read $o$. It is challenging to identify and record the source of the happens-before edge, because $T_1$ continues executing during the pessimistic transition by $T_2$. An eligible source needs to be (1) after $T_1$’s write to $o$, in order to capture the cross-thread dependence soundly, but (2) no later than $T_1$’s current execution point $e_1$, or else replay could deadlock: suppose $T_2$ records a future execution point $e_2$, and $T_1$ writes to $o$ again (not shown) between $e_1$ and $e_2$. $T_1$ would record an execution point after $T_2$’s read of $o$ as the source of another happens-before edge, creating a cycle of dependences.

In contrast, an optimistic conflicting transition triggers coordination, as shown in Figure 3.7(b). $T_1$ stops to respond to $T_2$ at a safe point, providing an opportunity to record the happens-before source. The responding safe point satisfies both requirements for an eligible source.

![Figure 3.7: The challenge of recording pessimistic conflicting transitions.](image)
The hybrid recorder could record every pessimistic access, but they are frequent enough that recording each one would be expensive. Alternatively, incrementing a counter at every pessimistic access would be efficient—but the replayed run would not know which accesses had been pessimistic versus optimistic during the recorded run. We encountered these challenges in our initial design of the hybrid recorder (Section 3.2.1). Our best attempted solution, which involved T2 writing into T1’s log for the example in Figure 3.7(a), performed worse on average than the optimistic recorder due to the cost of synchronizing on per-thread logs.

**Utilizing deferred unlocking.** These challenges are naturally addressed by, and thus motivate the use of, deferred unlocking (Section 3.2.1). By deferring unlocking of pessimistic states until program synchronization release operations (PSROs), the potential sources of happens-before edges are effectively limited.

The hybrid recorder handles pessimistic uncontended transitions involving conflicting states as follows. In both recorded and replayed executions, instrumentation at every PSRO and responding safe point increments a per-thread release counter. Using Figure 3.2(a) from Section 3.2.1 as an example, T1 increments its release counter before it releases the program lock m. When T2 changes the state to RdExR, it records the happens-before edge in its log by reading T1’s release counter and recording its value. Since each PSRO and responding safe point has release semantics, and each state change has acquire semantics, T2 is guaranteed to read a value of T1’s release counter that is at least as great as the value at the first PSRO after T1 writes to o. In addition, T2 cannot read a value that T1’s release counter has not reached, preventing deadlock during replay. During replay, T2 waits for T1’s release counter to reach the recorded value.
For a contended transition as in Figure 3.2(b), T2 initiates coordination. T1 unlocks o’s state to $\text{WrEx}^{\text{Pess}}_{T1}$, responds at a safe point, and records the response just as it would record an optimistic coordination response. T2 then records its uncontended transition from $\text{WrEx}^{\text{Pess}}_{T1}$ to $\text{RdEx}^{\text{RLock}}_{T2}$ as described above.

### 3.4 Enforcing Region Serializability

This section applies the hybrid state model to enforcing serializability (atomicity) of executed code regions, demonstrating how the model enables controlling cross-thread dependences.

#### 3.4.1 Optimistic RS Enforcer

Modern language memory models make strong guarantees for data-race-free (DRF) executions but provide virtually no guarantees for racy executions [3, 5, 20, 21, 92]. Adve, Boehm, and Ceze et al. have argued that languages and hardware must provide stronger memory models to avoid impossibly complex semantics [3, 20, 35]. Prior work enforces memory models that provide region serializability (RS) even for executions with data races [109,125]. We focus on work that introduces a memory model called statically bounded region serializability (SBRS) that provides serializability of regions that are bounded by program synchronization operations, method calls, and loop back edges [125].

Prior work, which we call the optimistic enforcer, enforces SBRS using optimistic tracking at each object access [125]. The optimistic enforcer provides region serializability via two-phase locking: each object access uses optimistic tracking to change the state if needed, and a region does not relinquish objects’ states (i.e., does not respond to coordination requests) until the region ends. However, to avoid deadlock, a thread may respond to coordination requests while itself waiting to complete a transition (lines 9 and 18 in Figure 3.1...
from Section 3.1.2), relinquishing ownership of objects’ states and thus potentially violating serializability.

The optimistic enforcer transforms regions at compile time so they can restart safely after responding to a coordination request.

### 3.4.2 Hybrid RS Enforcer

To understand the challenges of using hybrid tracking for the RS enforcer, consider how an RS enforcer based on *pessimistic tracking* would work. To preserve serializability, no pessimistic state locked during a region’s execution should be unlocked until the region completes. At region end, instrumentation should unlock each pessimistic state locked during the region’s execution.

However, using *hybrid* tracking presents a challenge, as illustrated in Figure 3.8. The compiler cannot predict whether the accesses to objects $o$ and $p$ will use pessimistic versus optimistic tracking, so each region end needs conditional checks for which pessimistic states to unlock, if any. Since we expect most accesses to be optimistic, most regions would need to unlock *no* pessimistic states. As statically bounded regions are short, the overhead of
checking at the end of each region would be significant. We encountered these challenges in our initial design of a hybrid enforcer (Section 3.2.1).

**Using deferred unlocking.** Our hybrid enforcer relies on deferred unlocking to address these challenges. Hybrid tracking defers unlocking of pessimistic states until program synchronization release operations (PSROs). PSROs are generally infrequent compared to region boundaries, so it is inexpensive to flush the lock buffer at each PSRO. Regions thus unlock pessimistic states only at region boundaries, preserving SBRS.

The one exception is pessimistic *contended* transitions, which trigger coordination in the middle of a region. Since the thread initiating coordination can respond to other threads’ coordination requests (line 39 in Figure 3.5), a region restarts after completing coordination, just as it does for optimistic conflicting transitions.

### 3.5 Adaptive Policy

This section addresses how to choose between pessimistic and optimistic states at run time. We introduce a *cost–benefit model* for deciding whether an object should be in pessimistic or optimistic states, and an efficient *policy* that approximates the cost–benefit model based on online profiling.

#### 3.5.1 Cost–Benefit Model

The basic idea of the cost–benefit model is that an object’s state should be pessimistic (versus optimistic) if and only if the total time incurred on optimistic transitions for the object would exceed the total time incurred on pessimistic transitions.

A limitation of our cost–benefit model is that it models pessimistic transitions based on pessimistic tracking *without deferred unlocking*. Thus, the model assumes that all accesses to
objects in optimistic states that trigger conflicting transitions (and thus coordination), would trigger uncontended (and thus coordination-free), non-reentrant pessimistic transitions if the objects were in pessimistic states.

The cost–benefit model considers each object individually. Let $N_{pess}$ be the number of pessimistic transitions that would occur for the object if its state were always pessimistic. $N_{pess}$ thus counts all program accesses to an object. Let $N_{conf}$ and $N_{nonConf}$ be the numbers of conflicting and non-conflicting transitions, respectively, that would occur if the state were optimistic. Since together $N_{conf}$ and $N_{nonConf}$ count all accesses,

$$N_{pess} = N_{nonConf} + N_{conf} \quad (3.1)$$

Let $T_{nonConf}$, $T_{conf}$, and $T_{pess}$ be the average time costs for an optimistic non-conflicting, optimistic conflicting, and pessimistic transition, respectively. The model considers these values to be (platform-specific) constants computed ahead of time, e.g., from the table in Section 3.1.2. An object’s state should be optimistic if and only if the following is true:

$$T_{pess} \times N_{pess} \geq T_{nonConf} \times N_{nonConf} + T_{conf} \times N_{conf} \quad (3.2)$$

The left-hand side of (3.2) is the total time spent on state transitions if the object’s state were pessimistic. The right-hand side is the total time on state transitions if the state were optimistic.

Applying (3.1) into (3.2) and transforming it yields:

$$N_{nonConf} \geq K_{conf} \times N_{conf} \quad (3.3)$$

3The model computes the time for non-conflicting transitions as simply the time for same-state transitions, ignoring other non-conflicting transitions (upgrading and fence transitions), which each incur a cost similar to a pessimistic transition’s cost.

4$T_{conf}$ is the time for a conflicting transition using explicit coordination.
where $K_{\text{conf}}$ is a run-time constant:

$$K_{\text{conf}} = \frac{T_{\text{conf}} - T_{\text{pess}}}{T_{\text{pess}} - T_{\text{nonConf}}}$$

Thus, according to (3.3), using the cost–benefit model requires knowing only the numbers of non-conflicting and conflicting transitions ($N_{\text{nonConf}}$ and $N_{\text{conf}}$), or merely their ratio.

### 3.5.2 Profile-Guided Adaptive Policy

Using the cost–benefit model to change each object’s state to optimistic or pessimistic at run time presents several challenges that we address as follows.

**Predicting the future.** The cost–benefit model seems to require oracle knowledge: it needs to know the future ratio $N_{\text{nonConf}}/N_{\text{conf}}$ when allocating an object, to initialize its state. The adaptive policy instead uses *online profiling*, assuming future behavior approximates past behavior in the same execution. Each object newly allocated by thread $T$ starts in the $\text{WrEx}^\text{Opt}_T$ state.

Profiling each object separately might limit the adaptive policy’s effectiveness. For example, if many objects each trigger only a few conflicting transitions, the policy will not transfer them to pessimistic states early enough. Profiling objects in *aggregate* (e.g., by object type) could enable allocating certain objects directly into pessimistic states. However, for our evaluated workloads, our policy gets nearly all of the possible benefit (Section 3.6.3).

**Efficient profiling.** Counting optimistic same-state transitions would be expensive because they are common (by design). The profiling thus counts only conflicting transitions for optimistic objects,\(^5\) but it counts all pessimistic transitions, since they are relatively

\(^5\)The policy counts only transitions that use explicit coordination, since implicit coordination is roughly as expensive as a pessimistic transition.
infrequent (by design). This policy thus readily transfers potentially high-conflict objects to pessimistic states—at which point more-intrusive profiling categorizes every pessimistic transition in order to determine whether an object should stay in pessimistic states or change back to optimistic states.

For each object o, the profiling counts the number of optimistic conflicting transitions \( o.numConflicts \). If an object experiences “enough” conflicting transitions, i.e., if

\[
o.numConflicts \geq Cutoff_{\text{conf}}
\]  

(3.4)

then the policy transitions the object to a pessimistic state.

For every pessimistic transition, profiling counts whether it was non-conflicting or conflicting. The policy changes an object back to optimistic based on the following formula, derived from (3.3):

\[
N_{\text{nonConf}} \geq K_{\text{conf}} \times N_{\text{conf}} + \text{Inertia}
\]  

(3.5)

The parameter \( \text{Inertia} \) avoids prematurely changing back to optimistic states before a significant amount of profiling has occurred.

Note that (3.4) and (3.5) are the definitions of \( \text{AdaptivePolicy.toPess} \) and \( \text{AdaptivePolicy.toOpt} \), respectively, used in Figure 3.5 (page 30).

Online profiling evaluates (3.5) efficiently at run time using one integer per object as follows. The first transition to pessimistic assigns \( \text{Inertia} \) to the per-object integer. Each pessimistic transition that would be conflicting adds \( K_{\text{conf}} \) to the integer. Each pessimistic transition that would be non-conflicting subtracts 1 from the integer. If the value becomes negative, then the policy transitions the object to an optimistic state when the state is next unlocked.
Checks and balances. By using a low value for $Cutoff_{confl}$, the adaptive policy quickly transitions objects to pessimistic states if they might be better off in pessimistic states, based on (3.4). Then profile-guided decisions based on (3.5) can more accurately distinguish objects that should be in pessimistic versus optimistic states. We find that supporting transitions to pessimistic using (3.4), without also supporting transitions back to optimistic based on (3.5), works poorly in general, regardless of $Cutoff_{confl}$’s value. To avoid repeatedly switching an object between optimistic and pessimistic states that should ideally remain optimistic, the policy disallows repeated transitions to pessimistic: each object starts in $WrEx^{Opt}_{T}$ state; it can transition to pessimistic and later can transition back to optimistic; after that, it must stay optimistic. Alternatively, the policy could allow repeated transitions from optimistic to pessimistic, but with a greater $Cutoff_{confl}$ value.

3.6 Evaluation

This section evaluates the run-time characteristics and performance of hybrid tracking, compared with pessimistic and optimistic tracking alone. It also compares the performance of the hybrid and optimistic versions of the dependence recorder and RS enforcer.

3.6.1 Implementation

We have implemented the hybrid state model, adaptive policy, hybrid dependence recorder and replayer, and hybrid RS enforcer in Jikes RVM 3.1.3, a Java virtual machine [9] that performs competitively with commercial JVMs [16]. We have made our implementation, which targets the IA-32 platform, publicly available on the Jikes RVM Research Archive. Our implementation builds on publicly available implementations of pessimistic
and optimistic tracking [25], the optimistic recorder and replayer [24], and the optimistic RS enforcer [125].

Jikes RVM’s dynamic just-in-time compilers insert instrumentation before every memory access, PSRO, and safe point in the application and Java libraries. The implementation adds two 32-bit words to each (scalar and array) object and static field: one for last-access state and another for the adaptive policy’s profile information. For exclusive states ($\text{WrEx}^*_T$ and $\text{RdEx}^*_T$), the state word encodes $T$’s (8-byte-aligned) address and uses remaining bits to differentiate states (e.g., pessimistic versus optimistic; $\text{WrEx}$ versus $\text{RdEx}$). For $\text{RdSh}^c_c$ states, the bits encode $c$ and the read-lock count, and differentiate pessimistic versus optimistic.

By targeting a managed language, our implementation can piggyback on existing language implementation features. Notably, coordination piggybacks on the safe point mechanism that commonly exists in managed language implementations. Our implementation modifies Jikes RVM’s object model and global statics table to add the metadata words. An implementation for a native language would need to add support for safe points, and could use shadow memory [104] to maintain metadata for shared-memory variables.

**Extraneous contention.** Due to limited bit patterns available in a metadata word, our prototype implementation omits the $\text{WrEx}^\text{RLock}_T$ state: a read to a $\text{WrEx}^\text{Pess}_T$ object triggers a transition to $\text{WrEx}^\text{WLock}_T$. The implementation could avoid this limitation with more

---

6http://sourceforge.net/p/jikesrvm/research-archive/43/
7http://sourceforge.net/p/jikesrvm/research-archive/49/
8http://sourceforge.net/p/jikesrvm/research-archive/48/
9The implementation could also instrument JNI callbacks that access Java’s heap, but currently it omits such instrumentation because the benchmarks we use do not invoke JNI callbacks.
engineering effort, e.g., by encoding an identifier for T, rather than T’s address, for $\text{WrEx}^\text{Pess}_T$ and $\text{RdEx}^\text{Pess}_T$ states.

Thus, the implementation may encounter pessimistic contention even in the absence of object-level data races. Suppose $T_1$ reads an object in $\text{WrEx}^\text{Pess}_{T_1}$ state, transitioning the state to $\text{WrEx}^{\text{WLock}}_{T_1}$. $T_2$ then reads the object, triggering a pessimistic contended transition. However, $T_1$ has only read the object since its last PSRO, i.e., no object-level data race exists in this case.

To measure potential costs incurred by triggering unnecessary coordination, we implemented and evaluated an alternate configuration in which a read of a $\text{WrEx}^\text{Pess}_{T_1}$ object by $T_1$ triggers a transition to $\text{RdEx}^{\text{RLock}}_{T_1}$. This configuration triggers coordination only when object-level data races exist, but it loses information about $T_1$’s previous write to the object, making it unsuitable for runtime support that needs to detect cross-thread dependences soundly. This unsound configuration provided no performance benefit, indicating that the default configuration is not encountering significant spurious contention in our experiments.

**Optimistic tracking performance issue.** When investigating the performance of high-conflict microbenchmarks (Section 3.6.5), we discovered an optimization opportunity that improves the performance of the optimistic tracking implementation. In particular, releasing a program lock that is a so-called “fat” lock [12] can incur significant latency, so making this operation a blocking safe point improves performance significantly. Our experiments do not include this optimization.
Table 3.3: The total number of spawned threads and maximum number of live threads for each program.

<table>
<thead>
<tr>
<th>Program</th>
<th>Total threads</th>
<th>Max live threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>eclipse6</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>hsqldb6</td>
<td>402</td>
<td>102</td>
</tr>
<tr>
<td>lusearch6</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>xalan6</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>avrora9</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>jython9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>luindex9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>luserarch9</td>
<td># cores</td>
<td># cores</td>
</tr>
<tr>
<td>pmd9</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>sunflow9</td>
<td># cores×2</td>
<td># cores</td>
</tr>
<tr>
<td>xalan9</td>
<td># cores</td>
<td># cores</td>
</tr>
<tr>
<td>pjbb2000</td>
<td>37</td>
<td>9</td>
</tr>
<tr>
<td>pjbb2005</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

3.6.2 Methodology

Our experiments execute benchmarked versions of real applications: the DaCapo benchmarks, versions 2006-10-MR2 and 9.12-bach (2009) [17] (limited to multithreaded programs that Jikes RVM can run), and fixed-workload versions of SPECjbb2000 and 2005.¹⁰

Table 3.3 shows the run-time threading behavior of each program. The first column is the total number of threads that each program spawns during execution, and the second column is the maximum number of threads that are simultaneously running at any instant. Some programs create threads according to the number of available processing units (cores), which is 32 in our experiments.

The experiments run on a system with four Intel Xeon E5-4620 8-core processors (32 cores total) running Linux 2.6.32. We build a high-performance configuration (Fast-AdaptiveGenlmmix) of Jikes RVM. Each performance result is the median of 20 trial runs; we also show the mean as the center of 95% confidence intervals. Each reported statistic is the mean from five statistics-gathering runs. We find that the run-to-run variations for statistics are insignificant.

To measure the state transition costs in CPU cycles for pessimistic and optimistic tracking (Section 3.1.2), the implementation uses the RDTSC instruction. The experiment divides the total cycles for each type of state transitions by the number of the state transitions. We run separate experiment for each type of state transitions, in order to avoid interference when instrumenting multiple types of state transitions.

3.6.3 Adaptive Policy Limit Study

To evaluate whether per-object profiling identifies most optimistic conflicting transitions in advance, we perform a limit study on optimistic tracking alone. Figure 3.9 plots a cumulative distribution of the number of optimistic conflicting transitions (explicit coordination only) triggered by each object. For each point \((x, y)\), \(y\) counts total conflicting transitions—as a percentage of all accesses—involving objects that have (so far) triggered at most \(x\) conflicting transitions. For example, \((4, 0.05\%)\) means that 0.05% of all accesses triggered conflicting transitions that were the first, second, third, or fourth conflicting transition triggered by the accessed object. The maximum \(y\) value for each program is its overall rate of conflicting transitions (explicit coordination only).

The plot shows that, at least for these programs, each object’s first few conflicting transitions together constitute an insignificant fraction of overall program accesses. For
Figure 3.9: Cumulative distribution of conflicting transitions (explicit coordination only) triggered per object for optimistic tracking. Both axes use a logarithmic scale. The legend sorts programs by their maximum y-axis value. Three programs have a conflict rate $<0.0001\%$ and are excluded.

For high-conflict programs, most conflicting transitions are to objects that have triggered many conflicting transitions (avrora9 is an exception). For low-conflict programs, the overall conflict rate is low, so conflicting transitions are negligible. Thus, per-object profiling can “catch” most conflicting accesses, leaving little additional opportunity for aggregate profiling.

The rest of the experiments use the following adaptive policy parameter values: $Cutoff_{conf} = 4$, $K_{conf} = 200$, $Inertia = 100$. We find that larger values of $Cutoff_{conf}$ have little impact except for avrora9, as Figure 3.9 would suggest. Performance is not very sensitive to the
other parameters; various values for $K_{conf}$ (20–1,600) and $Inertia$ (20–1,600) are effective. Section 3.6.7 provides more details.

### 3.6.4 Run-Time Characteristics

Table 3.4 counts state transitions under hybrid tracking. The table breaks down *Optimistic transitions* into *Same state* and *Conflicting* transitions, which have significantly different costs (Section 3.1.2). For comparison, transitions triggered under optimistic tracking alone are shown in parentheses.

The *Conflicting* column measures how well the adaptive policy achieves its primary goal of reducing conflicting transitions. The reduction is substantial for high-conflict programs: 43–98% for hsqldb6, xalan6, avror9, pmd9, xalan9, and pjbb2005. Hybrid tracking provides little or no improvement for low-conflict programs—but they incur low coordination costs anyway.

The *Same state* column measures the downside of transitioning to pessimistic states: some transitions that *would* have been optimistic same-state become pessimistic. Only a small fraction of same-state transitions become pessimistic, because the adaptive policy identifies pessimistic objects to transition back to optimistic states, based on accurate profiling of pessimistic objects.

As the table shows, the adaptive policy causes more same-state than conflicting transitions to become pessimistic (compared with optimistic tracking alone). However, this result does not imply a performance loss, since a conflicting transition costs 2–3 orders of magnitude more than a same-state transition. For these programs at least, the adaptive policy achieves its goal of eliminating most of the conflicting transitions—and thus most of the expensive coordination overhead—while minimizing pessimistic transitions.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Same state</td>
<td>Conflicting</td>
<td>Uncontended</td>
<td>%Reentrant</td>
<td>Contended</td>
<td></td>
</tr>
<tr>
<td>eclipse6</td>
<td>(1.2×10^{10})</td>
<td>1.2×10^{10}</td>
<td>(1.3×10^{5})</td>
<td>1.3×10^{5}</td>
<td>1.5×10^{8}</td>
<td>32%</td>
</tr>
<tr>
<td>hsqldb6</td>
<td>(6.1×10^{8})</td>
<td>6.1×10^{8}</td>
<td>(9.2×10^{5})</td>
<td>5.2×10^{5}</td>
<td>4.7×10^{8}</td>
<td>64%</td>
</tr>
<tr>
<td>lusearch6</td>
<td>(2.4×10^{9})</td>
<td>2.3×10^{9}</td>
<td>(4.4×10^{3})</td>
<td>4.3×10^{3}</td>
<td>2.6×10^{2}</td>
<td>30%</td>
</tr>
<tr>
<td>xalan6</td>
<td>(1.1×10^{10})</td>
<td>1.0×10^{10}</td>
<td>(1.8×10^{7})</td>
<td>3.9×10^{5}</td>
<td>2.1×10^{8}</td>
<td>52%</td>
</tr>
<tr>
<td>avrora9</td>
<td>(6.0×10^{6})</td>
<td>6.0×10^{6}</td>
<td>(6.0×10^{6})</td>
<td>2.7×10^{6}</td>
<td>8.4×10^{6}</td>
<td>17%</td>
</tr>
<tr>
<td>jython9</td>
<td>(5.1×10^{9})</td>
<td>5.1×10^{9}</td>
<td>(6.7×10^{1})</td>
<td>7.3×10^{1}</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>luindex9</td>
<td>(3.4×10^{8})</td>
<td>3.4×10^{8}</td>
<td>(3.7×10^{2})</td>
<td>3.8×10^{2}</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>lusearch9</td>
<td>(2.3×10^{9})</td>
<td>2.3×10^{9}</td>
<td>(2.8×10^{3})</td>
<td>2.3×10^{3}</td>
<td>3.9×10^{3}</td>
<td>44%</td>
</tr>
<tr>
<td>pmd9</td>
<td>(5.6×10^{8})</td>
<td>5.5×10^{8}</td>
<td>(4.2×10^{4})</td>
<td>1.7×10^{4}</td>
<td>1.9×10^{5}</td>
<td>58%</td>
</tr>
<tr>
<td>sunflow9</td>
<td>(1.7×10^{10})</td>
<td>1.7×10^{10}</td>
<td>(6.1×10^{5})</td>
<td>6.2×10^{5}</td>
<td>5.9×10^{5}</td>
<td>92%</td>
</tr>
<tr>
<td>xalan9</td>
<td>(1.0×10^{10})</td>
<td>9.8×10^{9}</td>
<td>(1.7×10^{7})</td>
<td>2.9×10^{5}</td>
<td>1.9×10^{8}</td>
<td>68%</td>
</tr>
<tr>
<td>pjbb2000</td>
<td>(1.7×10^{9})</td>
<td>1.7×10^{9}</td>
<td>(9.5×10^{5})</td>
<td>9.3×10^{5}</td>
<td>2.4×10^{9}</td>
<td>58%</td>
</tr>
<tr>
<td>pjbb2005</td>
<td>(6.6×10^{9})</td>
<td>6.5×10^{9}</td>
<td>(4.4×10^{7})</td>
<td>8.4×10^{5}</td>
<td>1.4×10^{8}</td>
<td>32%</td>
</tr>
</tbody>
</table>

Table 3.4: State transitions for hybrid tracking, compared with state transitions for optimistic tracking alone (shown in parentheses).
The *Pessimistic* columns show the number of pessimistic transitions under hybrid tracking. We note that deferred unlocking enables a significant fraction of *Uncontended* accesses to be *Reentrant* and thus avoid atomic operations. Still, a substantial fraction of pessimistic accesses require atomic operations, so pessimistic tracking alone would be costly even if it used deferred unlocking.

For most programs, a small fraction of pessimistic accesses are *Contended*, indicating that deferring unlocking of pessimistic states is generally successful. However, for *avrora9* and *pjbb2005*, contended transitions are of the same order as optimistic conflicting transitions, so hybrid tracking still incurs a considerable amount of coordination. Investigating further, we find that the contention is, as expected, due to object-level data races. In *pjbb2005*, contention is caused by true (precise) data races. In *avrora9*, contention is caused by both true and false (object-level-only) data races.

The last two columns show transitions between pessimistic and optimistic states. Not all of the objects that transition from optimistic to pessimistic should ideally be pessimistic. The fraction of pessimistic objects transitioned *back* to optimistic states varies significantly across the programs but is often substantial, indicating that accurate profiling of pessimistic objects is crucial.

### 3.6.5 Performance of Tracking Alone

Figure 3.10 compares the performance of hybrid tracking with pessimistic and optimistic tracking alone (no runtime support on top of dependence tracking). Each bar shows the run-time overhead added over unmodified Jikes RVM. For *sunflow9*, the mean overhead is noticeably higher than the median for several configurations. Across many additional trials, we found that about 15% of the trials run substantially slower than the rest of the trials.
Figure 3.10: Run-time overhead of pessimistic and optimistic tracking, compared with hybrid tracking. Each bar is the median of 20 trials. The intervals are 95% confidence intervals centered at the mean. Overheads exceeding 120% are labeled using two significant figures.

_Pessimistic tracking_ adds 340% overhead on average (excluding sunflow9, the geomean is 210%), showing that pessimistic states must be applied judiciously. In contrast, the average overhead of _Optimistic tracking_ is just 28%, but a few high-conflict programs (xalan6 and pjbb2005) incur substantially higher costs.

_Hybrid tracking w/infinite cutoff_ uses hybrid tracking but sets \(C_{\text{confl}}\) to \(\infty\), so no object ever transitions to pessimistic states. This configuration measures only the costs, not the benefits, of hybrid tracking over optimistic tracking. The average cost over optimistic tracking is 2.3% (of baseline execution time).

_Hybrid tracking_ uses the default values of \(C_{\text{confl}}\) and other parameters. Hybrid tracking significantly improves the performance of several programs that perform poorly with optimistic tracking—the same programs that have many conflicting transitions reduced by the adaptive policy (Table 3.4). Hybrid tracking reduces overhead by 63% (65% \(\rightarrow\) 24%) for xalan6; by 74% (19% \(\rightarrow\) 5%) for xalan9; and by 45% (from 110% \(\rightarrow\) 49%) for pjbb2005. Despite reducing conflicting transitions significantly for hsqldb6 (Table 3.4),
hybrid tracking has little performance impact because hsqldb6’s conflicting transitions mainly use implicit coordination, which costs about as much as a pessimistic transition.

*Ideal* is the overhead of optimistic tracking, but *without performing coordination* for conflicting transitions. This *unsound* configuration estimates the cost of all conflicting transitions becoming pessimistic and all same-state transitions remaining optimistic. It adds 14% on average, representing an estimated upper bound on the performance that hybrid tracking might be able to provide.

Hybrid tracking adds 22% average overhead, 21% less than optimistic tracking’s 28% overhead. Hybrid tracking incurs 27% less overhead than *Hybrid tracking w/ infinite cutoff*, recovering most of the overhead difference between optimistic tracking alone and the ideal, unsound configuration.

While optimistic tracking provides the best performance for low-conflict programs, hybrid tracking provides better performance for high-conflict programs. On average, hybrid tracking adds lower overhead than both pessimistic and optimistic tracking alone.

Many of the programs we evaluate perform relatively little shared-memory communication [71]. These programs may or may not accurately represent all real-world parallel programs in the wild. Because of these programs’ low average communication, optimistic tracking performs well on average, leaving little room for hybrid tracking to improve. Nevertheless, only hybrid tracking can scale to diverse communication patterns: it helps cases for which optimistic tracking performs poorly, without harming cases for which optimistic tracking performs well.
Stress tests. In addition to large, real programs, we evaluate pessimistic, optimistic, and hybrid tracking on two microbenchmarks—one well synchronized and one with data races—that represent extreme, high-conflict cases. Each microbenchmark spawns eight threads; each thread repeatedly increments a global counter in a loop. Figure 3.11 shows, for each microbenchmark, the code executed by each thread, as well as run-time overhead over execution time on the unmodified JVM. The program `syncInc` acquires a global lock before every increment, whereas `racyInc` does not.

The figure shows that for `syncInc`, hybrid tracking significantly reduces overhead relative to optimistic tracking (84% versus 1200%), eliminating most coordination thanks to object-level data race freedom. For this program, hybrid tracking essentially mimics pessimistic tracking by using pessimistic transitions. However, hybrid tracking incurs more overhead in order to defer unlocking states and to perform profiling.

Figure 3.11: Run-time overhead of tracking alone on microbenchmarks.
In contrast, \texttt{racyInc} represents a worst case for hybrid tracking since almost all conflicting accesses are involved in data races. Hybrid tracking adds 4300\% overhead because threads repeatedly trigger coordination in order to perform pessimistic contended transitions. Upon further investigation, we find that although only 24\% of memory accesses perform pessimistic contended transitions, most of these accesses trigger coordination more than once. Hybrid tracking could alleviate this deficiency by modifying the adaptive policy to switch a pessimistic object back to optimistic states if accesses to it trigger coordination frequently.

Pessimistic and optimistic tracking both add about 1200\% overhead for \texttt{racyInc}; this similarity is initially surprising considering that \texttt{racyInc} executes many conflicting accesses, which are typically more expensive for optimistic tracking than for pessimistic tracking. We find that in optimistic tracking, only 8.5\% of all accesses trigger conflicting transitions, because a thread that locks a state can perform several same-state transitions before another thread initiates a conflicting transition. In contrast, in pessimistic tracking, another thread tries to lock a state more quickly, leading to more remote cache misses: 26\% of pessimistic tracking’s accesses lock a state with a different thread than the previous access.
3.6.6 Performance of Runtime Support

This section compares optimistic and hybrid versions of the dependence recorder and RS enforcer. We have not implemented or evaluated pessimistic runtime support, since pessimistic tracking alone is slower than both optimistic and hybrid runtime support.

**Dependence recorder.** Figure 3.12(a) shows the performance of the optimistic and hybrid dependence recorders and replayers. Hybrid tracking improves the recorder's performance significantly for the high-conflict programs xalan6, xalan9, and pjbb2005, and incurs modest overhead for low-conflict programs. On average it reduces overhead by 11% (from 46 to 41%). While the hybrid recorder triggers less coordination than the optimistic recorder, it still detects and records the same number of cross-thread dependences as the optimistic recorder does. This fact explains why the hybrid recorder's improvement over the optimistic recorder is smaller than for hybrid tracking over optimistic tracking alone.

The optimistic replayer is not fully robust: it successfully replays 11 out of 13 programs (failing on eclipse6 and xalan9) [24]. The optimistic replayer adds 20% overhead on average—lower than the optimistic recorder because it is cheaper to replay known dependences than record unknown dependences. The replayer outperforms the baseline substantially for pjbb2005. This result is not an experimental anomaly; the replayer elides program synchronization operations and replays only the recorded dependences, so it can outperform baseline execution for programs dominated by coarse-grained, overly conservative synchronization.

Our hybrid replayer successfully replays all 11 programs that the optimistic replayer can replay. The hybrid replayer adds 24% overhead on average, slower than the optimistic replayer, due to the cost of maintaining the per-thread release counter, as well as the fact that
hybrid tracking cannot reduce the number of replayed cross-thread dependences. Overall, hybrid tracking improves record time and degrades replay time—a worthwhile tradeoff since (1) optimizing record is more important since it is usually slower than replay, and (2) replay performance is not important in all settings (e.g., offline replay).

Region serializability enforcer. Figure 3.12(b) shows the overhead of enforcing SBRS using optimistic versus hybrid tracking. The hybrid enforcer substantially improves the performance of xalan6, xalan9, and pjbb2005. This reduction is similar to the reduction between hybrid and optimistic tracking alone—which is unsurprising since the hybrid enforcer employs hybrid tracking in essentially the same way as the optimistic enforcer employs optimistic tracking. On average, the hybrid enforcer reduces overhead by 13% over the optimistic enforcer (from 39% to 34%).

The performance story for runtime support is similar to the story for dependence tracking alone: hybridizing pessimistic and optimistic tracking overcomes the limitations of both, providing the best overall performance for a mix of low- and high-conflict programs.

3.6.7 Sensitivity Analysis

This section evaluates the performance impact of different values of parameters for the adaptive policy (Section 3.5): $Cutoff_{conf}$, $K_{conf}$, and Inertia. This section’s results are based on 10 trials of each experiment.

For most programs, we do not detect that performance is sensitive to varying values of any of the three parameters. In particular, the 95% confidence intervals overlap. We report the only three cases in which the 95% confidence intervals do not overlap, i.e., we only show results for programs that have performance differences that are statistically significant.
Table 3.5: Run-time overhead of hybrid tracking with varying parameters for programs that showed statistically significant sensitivity. The default value and overhead are in bold.

Table 3.5(a) shows run-time overhead using different values of $Cutoff_{confl}$ for avrora9, the only program that shows statistically significant sensitivity to different $Cutoff_{confl}$ values. The table varies $Cutoff_{confl}$ from 2 to 256; $K_{confl}$ and Inertia are fixed at their defaults. Hybrid tracking achieves the best performance (36% overhead) for avrora9 with a $Cutoff_{confl}$ of 2. Increasing values of $Cutoff_{confl}$ gradually degrade performance. This result confirms what Figure 3.9 would suggest—that the bulk of conflicting transitions for avrora9 are among objects’ first 16 conflicting transitions. A low $Cutoff_{confl}$ value enables avoiding more optimistic conflicting transitions; $Cutoff_{confl} \geq 16$ misses most of this benefit.

Table 3.5(b) shows run-time overhead for values of $K_{confl}$ from 20 to 1600; other parameters are fixed at defaults. For xalan6, the only significantly sensitive program, a small $K_{confl}$ value is less effective than larger values. This pattern suggests that xalan6 has objects whose non-conflicting-to-conflicting ratio is around 20–50; hybrid tracking performs better if the adaptive policy keeps these objects in pessimistic states rather than changing them back to optimistic states.
Table 3.5(c) shows the run-time overhead of using different Inertia values (20–1600); other parameters are fixed at defaults. The result resembles Table 3.5(b)’s result. For xalan6, a small Inertia value is less effective than larger values, presumably because a small value causes some objects to prematurely switch back to optimistic states.

We note that for pjbb2005, increasing Inertia can effectively reduce the variability of the run time, although the mean and median are unaffected. We conclude that Inertia should be chosen to be reasonably large, on the order of $K_{conf}$, in order to serve its design purpose (Section 3.5.2).

### 3.7 Contributions and Impact

Hybrid tracking uses a hybrid state model and adaptive policy to combine pessimistic and optimistic tracking effectively and efficiently, achieving better average performance than either alone. We demonstrate hybrid tracking’s potential by building runtime support to record dependences and enforce region serializability. The results motivate hybrid tracking’s use in building efficient runtime support that targets diverse applications on commodity systems.

**Contributions.** The primary contributions of this work are as follows:

- Hybrid tracking demonstrates a novel direction to build sound and efficient runtime support that overcomes performance limitations of using either pessimistic or optimistic tracking alone.

- We show that combining pessimistic and optimistic tracking naively is insufficient for achieving sound and efficient runtime support, due to a fundamental mismatch
between them. Hybrid tracking effectively addresses this key challenge by introducing deferred unlocking of pessimistic states.

• We extend two kinds of runtime support, demonstrating how runtime support can use hybrid tracking to soundly detect or control cross-thread dependences.

• Thorough evaluation shows characteristics and potentials in several aspects of hybrid tracking, as well as inherent inter-thread communication patterns in popular benchmarks that were previously little studied.

High runtime overhead has been a critical obstacle to the widespread adoption of runtime support that soundly deals with racy executions. Hybrid tracking is a general framework that addresses the performance limitations of many analyses that require dependence tracking, demonstrating a new direction for building efficient, flexible, software-only runtime support that targets diverse parallel software systems. As a result, hybrid tracking promotes the adoption of runtime support into software development, testing, and production, which will in turn improve developers’ productivity and software’s reliability.
Chapter 4: RegPlay: Efficient, Software-Only Multithreaded Record & Replay

While hybrid tracking is a general framework that improves the performance of dependence tracking, it is possible for a specific runtime support to further optimize how dependence tracking is done based on the runtime support’s need. Tracking all dependences, which hybrid tracking does, could be excessive and overkill for analyses that do not need knowledge of all dependences.

This chapter demonstrates how to improve the performance dependence tracking in the context of multithreaded record & replay. We present RegPlay, a novel record & replay approach that does not need to track all dependences but still provides replay determinism.

4.1 Background and Motivation

Multithreaded programs in shared-memory systems are inherently nondeterministic. Memory accesses by different threads can interleave in many ways, leading to nondeterministic program behaviors. Nondeterminism considerably complicates detection and reproduction of software bugs, and hinders system designs that require replication of executing process.
**Record & replay for multithreaded programs.** Record & replay is an effective approach to eliminate nondeterminisms in multithreaded programs [10, 24, 37, 45, 58, 64–66, 68, 79–81, 85, 98, 102, 110, 112, 118, 137, 142, 143]. The basic idea is to make one execution record sufficient information about runtime events, such as memory accesses, synchronization operations, branching decisions, and values of I/O operations, so that another execution can faithfully replay the behaviors of the original execution. *Replay determinism* refers to the property that replayed execution behaves exactly the same as recorded execution. A record & replay approach is *sound* if it can always provide replay determinism.

A replayed execution can be *offline* or *online*. An offline execution runs after the original recorded execution completes, which is useful for reproducing errors and debugging. An online execution runs concurrently with the recorded execution, enabling replication-based fault tolerance [26] and offloading dynamic analyses to replicated processes [38, 105].

Despite decades of research effort, multithreaded record & replay still remains rarely deployed in production. This phenomenon is mainly due to two challenges: (a) the performance overhead to record complete information about thread interleavings or data dependences is prohibitive; (b) it is difficult and often impractical to eliminate all sources of nondeterminisms in production settings. Since it mostly takes system-wide engineering effort, rather than research innovations in language and software, to overcome the second challenge, this paper mainly targets at addressing the first challenge. Thus, unless stated otherwise, our algorithm assumes the external environment (such as I/O operations, return values of system calls, and memory allocations) is deterministic.

Existing research has reached two fundamental insights that serve as the basis for many multithreaded record & replay approaches:
• For a data-race-free (DRF) execution, enforcing the same order of synchronization operations is sufficient to ensure replay determinism.

• For any execution, reproducing all cross-thread data dependences (write–read, read–write, and write–write dependences involving two threads) ensures replay determinism.

Recording and replaying synchronization operations typically adds low runtime overhead because they are infrequent. However, many real-world program executions do contain data races. Enforcing synchronization order does not guarantee the same order for memory accesses involved in data races. Thus, solely relying on recording and replaying synchronization order is insufficient to provide practical, sound multithreaded record & replay.

On the other hand, recording all cross-thread dependences tends to incur formidable runtime overhead. This is due to various reasons. (1) Every memory access to potentially shared variable must be instrumented. (2) To ensure correctness, the instrumentation must add synchronization to provide atomicity of the instrumentation and associated program access. (3) At a program read, the instrumentation usually introduces additional writes to memory due to added synchronization, or in order to track read–write dependences, leading to additional cache misses for mostly read-shared accesses. (4) Many recorded dependences could be unnecessary, because they are transitively implied by other dependences or ordering constraints.

Prior work has either made tradeoff between soundness of replay determinism and record efficiency, or provided replay determinism but introduced drawbacks such as high recording overhead, low scalability, inability to support both offline and online replay, and reliance on unrealistic hardware. Approaches that completely ignore data races are unsound for racy executions [58, 118]. Some approaches sidestep the challenge of tracking racy
accesses but introduce serious limitations [10, 68, 81, 110, 137, 142]. Other approaches track dependences directly but either incur high overhead, or suffer from other limitations mentioned above [24, 45, 66, 76, 79, 80, 85, 144]. Approaches that rely on custom hardware support are impractical for commodity software systems [64, 65, 98, 102, 112, 143].

Two recent software-only record & replay approaches have made notable performance improvement over prior dependence-tracking approaches. The first approach utilizes an optimistic tracking mechanism called Octet to achieve good average performance for the recorded execution [24, 25]. Octet requires no synchronization for accesses not involved in cross-thread dependences, but incurs significant coordination overhead for accesses that are conflicting (i.e., they are involved in cross-thread dependences)—even if the conflicting accesses are well synchronized. As a result, for programs with only a few conflicting accesses (e.g., 0.1% of all accesses), their dependence recorder incurs high overhead (e.g., > 100%) [24, 25].

Another noteworthy shortcoming of the Octet-based dependence recorder is that it records dependences many more than the minimal number that is sufficient to provide replay determinism. In particular, it would record a dependence even though the dependence is transitively implied by another dependence. Figure 4.1 shows such an example. The recorder would record both write–read dependences on variables $x$ and $y$, although recording and replaying only the dependence for $y$ is sufficient for replay determinism. As a result of this shortcoming, despite the authors’ follow-up effort to reduce the amount or cost coordination for conflicting accesses, the performance improvement on the dependence recorder is limited [34, 148]

The second approach, called Light, achieves low recording overhead by recording only intra-thread and cross-thread write–read dependences [85]. Light requires running an
Figure 4.1: A transitively implied cross-thread dependence. The write–read dependence on \( x \) is transitively implied by the write–read dependence on \( y \).

offline constraint solver to generate a feasible replay schedule that satisfies the recorded dependences. The reliance on the constraint solver significantly limits the usability and scalability of Light. Light can not support online replay, and it is expensive to scale for large, long-running programs with many memory accesses. Besides, Light is likely to record more dependences than necessary, because it records intra-thread write–read dependences and transitively implied dependences involving multiple variables (e.g., the case in Figure 4.1).

In addition, Light’s approach does not provide replay determinism in all cases. The authors have proved that preserving all write–read dependences alone is necessary and sufficient to reproduce most program bugs of interests [85]. However, Light does not guarantee a replayed execution to terminate in the same memory state as the recorded execution does. Prior work has shown that replayed executions should terminate in the same memory state in order to be useful under certain scenarios [81].

**Data races and region conflicts.** As Chapter 2 explains, data races are known to cause many problems, including nondeterminism [3, 22, 29, 32, 54, 72, 74, 103, 124, 128]. It is also impractical to fix all data races in order to build an efficient record & replay system. In the attempt to fix the issues of data races in weak memory models, researchers have proposed new memory or execution models in order to provide stronger semantics for racy
executions [7, 16, 35, 47, 90, 91, 94, 109, 123, 125–127, 131]. Most of these approaches try to enforce serializability of some type of code regions. Some of these models could simplify the design of record & replay, as recording and replaying the commit order of code regions would ensure replay determinism. However, as none of these stronger models is actually employed in production software, record & replay systems that rely on these models are also unrealistic.

A recent work called Valor provides region conflict exception execution model at low overhead. A region conflict occurs when two regions (e.g., SFRs) containing conflicting accesses execute simultaneously, potentially violating the serializability of regions. Every SFR conflict corresponds to a data race, but a data race may not trigger an SFR conflict. Valor throws an exceptions for each detected region conflict, providing serializability of SFRs for executions that do not trigger region conflict exceptions [16].

Valor’s key insight to achieve low overhead is that detecting read–write conflicts is expensive, and it is unnecessary to eagerly track the last reader of shared variables for region conflict detection. Valor’s approach of reasoning racy executions via detection of region conflicts, and its insight for achieving low runtime overhead, have inspired our record & replay algorithm.

Existing multithreaded record & replay approaches either incur heavy runtime overhead, or suffer from inherent limitations. Lessons from these approaches, as well as recent advancement in region conflict detection, have motivated our novel record & replay algorithm. Our record & replay algorithm aims to provide replay determinism for all executions at low recording overhead, while support both offline and online replay without using custom hardware.
4.2 Design

This section presents RegPlay, a novel multithreaded record & replay algorithm. To achieve low recording overhead, RegPlay employs lock-free instrumentation that only records write–read and write–write dependences that are not transitively implied by other dependences between two threads. RegPlay’s replay algorithm enforces the recorded write–read and write–write dependences directly, while detects and resolves violations of read–write dependences, ensuring replay determinism.

4.2.1 Overview

RegPlay is influenced by prior work for fast region conflict detection [16]. Similar to the definition in prior work, a region can be a code sequence bounded by either program synchronization operations (synchronization-free region, SFR), or only program synchronization release operations (release-free region, RFR). Nonetheless, RFR is more efficient and has the same capability as SFR for detecting problematic data races [16].

At a high level, RegPlay maintains a per-thread clock that is incremented at every region boundary during both record and replay executions, in order to identify the current epoch at every access. An epoch is a tuple of a thread ID and a clock value, which identifies a dynamic instance of a region. RegPlay also maintains the epoch of last write access to every shared-memory variable in both record and replay. RegPlay uses per-thread log files to record and replay dependences.

At a read or write access, the record algorithm detects write–read and write–write dependences and if they are involved in region conflicts. Figure 4.2 shows all possible cases. In all cases, thread T1 performs the first write access, forming the source of an dependence edge. T2 performs the second read or write access, constituting the sink of the
dependence edge. A dashed line denotes a region boundary. Gray text summarizes action of instrumentation. $j$ and $j+1$ on the left of $T1$ denote the current clock value of $T1$. The current clock value of $T2$ is omitted. Initially, the variable $x$’s last write’s epoch is $p@T0$.

If there is no region conflict, $T2$ records a dependence edge from the end of first access’s region to the second access (Figure 4.2(a)). For a write–read dependence that triggers a region conflict, $T2$ records the value of the read (Figure 4.2(b)). For a write–write dependence involved in a region conflict, the two threads resolve the region conflict via coordination and $T2$ records it as if it is not involved in a region conflict (Figure 4.2(c)). The coordination involves $T2$ requests $T1$ to end its current region early, then $T1$ would end its region at its next safe point—a point that every thread would encounter in bounded amount of time, such as method entries, exits and loop back edges. $T1$ would increment its clock and record this safe point as a special region boundary. The record algorithm does not detect or record any read–write dependences.

The replay algorithm enforces a recorded write–write dependence and a region-conflict-free write–read dependence by making the second access wait for the first access’s region to end. For a write–read dependence involved in a region conflict, the read access would just return the recorded value.

Since read–write dependences are not recorded, it is possible that a read–write dependence gets “flipped” during replay, such that the write executes before the read and violates the original dependence. Figure 4.3 shows an example of a flipped read–write dependence. $v1$ and $v2$ denote the value for a write or read access. During record, the read–write dependence from $T2$’s read to $T1$’s second write to $x$ is not recorded (Figure 4.3(a)). During replay, $T1$’s second write executes before $T2$’s read, flipping the original read–write dependence
Figure 4.2: Overview of RegPlay’s record algorithm. (a) RegPlay records a dependence from end of a region to current access for a write–write or write–read dependence that does not constitute a region conflict. (b) RegPlay records the value of a read access if the read triggers a region conflict with last write. (c) For a write–write dependence involved in a region conflict, RegPlay resolves the region conflict via coordination.

(Figure 4.3(b)). RegPlay’s replay algorithm is able to detect such flipped read–write dependences, then the algorithm looks up for past values written to \( x \) and returns a correct one for the read. Detecting and recovering flipped read–write dependences is a key intellectual contribution of RegPlay’s replay algorithm, which we will discuss in details in Section 4.2.4.

The following three sections describe RegPlay’s record and replay algorithms in more details. Notably, RegPlay maintains two crucial data structures that are not mentioned above: (a) the dependence vector clock during both record and replay (Section 4.2.2), which helps avoid recording transitively implied dependences and detect flipped read–write dependences;
Figure 4.3: Example of a flipped read–write dependence in replay and how RegPlay handles it. (a) During record, the read–write dependence from T2’s read to T1’s second write is not recorded. (b) During replay, if the read–write dependence is flipped, RegPlay would detect this inconsistency and return a correct value at the read.

(b) the write log during replay (Section 4.2.4), which helps the replay algorithm to recover from flipped read–write dependences.

### 4.2.2 Common Data Structures

This section describes data structures and operations required by both the record and the replay algorithms. RegPlay maintains the following *per-thread* data structures:

**c**: A clock that is incremented at every region boundary. This clock is analogous to the logical clock used many dynamic analyses [16, 53, 97]. An *epoch* is a pair of a thread $t$ and a clock value $c$, denoted as $c@t$. An epoch uniquely identifies a dynamic instance.
of a region. For clarity of exposition, \texttt{clock(t)} returns thread \( t \)’s current clock value, and \texttt{epoch(t)} returns the epoch \( \text{c}@t \) of thread \( t \)’s ongoing region.

\textbf{K} : A \textit{dependence vector clock}, which maps each thread to a clock value. Unlike the classic vector clock for tracking happens-before relations [77, 97], the dependence vector clock tracks the order established by cross-thread write–read and write–write dependences. The value \( T.K[t] \) is the largest clock value of thread \( t \) that \( K \)’s owner thread \( T \) has observed from write–read and write–write dependences sourcing from thread \( t \). An invariant is that \( \text{clock}(T) = T.K[T] \).

\textbf{ypc} : A \textit{yield point counter} that is incremented at every method entry, method return, and loop back edge. RegPlay requires a mechanism for the record and the replay algorithms to agree on \textit{when} an event occurs. RegPlay borrows the design of \textit{dynamic program location} from prior work, which uses the triple of a thread identifier, a yield point counter value, and a static site (e.g., method and offset) to identify each dynamic instance of a static instruction [24].

\textbf{log} : A log file for storing events during record, or for loading recorded events during replay.

All above per-thread data structures except \( c \) are only accessed by their owner thread, avoiding additional synchronization. Only the owner thread of \( c \) can modify \( c \), but a thread may read a remote thread’s \( c \). Implementation only needs memory fences at certain places that access \( c \).

RegPlay also maintains the following metadata for each shared-memory variable:

\textbf{W}_x : The epoch \( (\text{c}@t) \) of the last region that writes to variable \( x \).
Algorithm 1 shows that at a region boundary, a thread $T$ increments its clock and the component for $T$ in its dependence vector clock.

Algorithm 1

REGION_END: thread $T$ at a region boundary

1: \begin{align*}
    T.c & \leftarrow \text{clock}(T) + 1 \\
2: \quad T.K[T] & \leftarrow \text{clock}(T)
\end{align*}

4.2.3 Record

Algorithms 2–5 show the analysis for RegPlay’s record algorithm. The algorithms use $T$ for the current thread, and $t$ or $t'$ for other threads. As stated earlier, the record algorithm only detects and records write–write and write–read dependences, but not read–write dependences. The record algorithm handles an access differently if it triggers a region conflict.

Read and write instrumentation. Algorithm 2 shows the analysis when thread $T$ writes to variable $x$. The analysis only executes if the write is the first write to $x$ in the current region, in which case the last write region would be different from the current region (line 1). Lines 3–9 try to atomically changes the metadata $W_x$ to the current region’s epoch, while determining if the write–write dependence should be recorded. On line 9, the atomic instruction \text{CAS}(addr, oldVal, newVal) tries to update addr from oldVal to newVal, returing true on success. If $T.K[t] < c$ (line 5), $T$ must have not observed other write–write or write–read dependences originating from the region $c@t$, thus the current write–write dependence must be recorded as it is not transitively implied by other write–write or write–read dependences from $t$ to $T$. The algorithm then checks if the region $c@t$ is still ongoing (line 7), and initiates coordination (Algorithm 4) to make sure the region $c@t$
has ended before the program write occurs, which effectively resolves the region conflict. Finally, T updates its dependence vector clock and records the dependence if needed. The first parameter of the T.log.record() method denotes the type of the recorded event.

Algorithm 3 illustrates the instrumentation before and after a thread T reads variable x. The original program read is at line 5, which returns the value v. The instrumentation first loads the metadata W_x (line 1). Similar to Algorithm 2, line 3 checks if the write–read dependence is implied by other write–write or write–read dependences from t to T, and further checks for region conflict if the write–read dependence is not implied. Note that if the last write is by thread T, T.K[T] must be greater than or equal to c. After the program read, the instrumentation reloads the metadata W_x if the read does not trigger a region conflict, and checks if the metadata has changed (line 6). If the metadata has changed, a remote thread must have performed a write to x concurrently with the read, so this read is also involved in a region conflict. For a read involved in a region conflict, T records the value of the read. Otherwise, T updates its dependence vector clock and records the write–read dependence if needed.

**Coordination.** In order to support the coordination used in Algorithm 2, each thread maintains two additional counters: req and reqSeen. Algorithms 4 and 5 show the details for the threads initiating coordination and responding to coordination, respectively. When a thread T calls coordinate(t, c), T first sends a request to t by atomically incrementing t.req. Then T waits until t ends its region identified by c@t. When thread t reaches a safe point, it checks if there are any pending requests by comparing its req and reqSeen counters. If yes, t updates the reqSeen counter to the value of req counter, and terminates its current region
and records this special region boundary. Such a special region boundary essentially splits a larger SFR or RFR region into two, in order to resolve a region conflict.

A safe point is a point that a thread can be reached in bounded amount of time, and that is not between a memory access and its instrumentation. Safe points can be method entries and returns, and loop back edges. Conveniently, managed language runtimes already have such points inserted (e.g., for garbage collection and adaptive optimizations). The coordination implementation can piggyback on such mechanism.

Our design of the coordination mechanism is influenced by similar approaches in prior work [25, 30, 34, 75, 119]. In order to ensure responsiveness and prevent deadlock, the coordination mechanism should also support implicit coordination, which happens when the remote thread is blocked. Prior work has designed similar mechanisms. Our implementation (Section 4.3) supports implicit coordination, but we omit it here as it is a straightforward extension.

### Algorithm 2

**WRITE [RECORD]: thread \( T \) writes variable \( x \) with value \( v \)

1: if \( W_x \neq \text{epoch}(T) \) then \( \triangleright \) First write in the region
2: let shouldRecord \( \leftarrow \) false
3: repeat
4: let \( c@t \leftarrow W_x \)
5: if \( T.K[t] < c \) then \( \triangleright \) This dependence is not implied
6: shouldRecord \( \leftarrow \) true
7: if \( c = \text{clock}(t) \) then \( \triangleright t's \ region \ is \ ongoing \)
8: coordinate(t, c) \( \triangleright \) Wait till \( t \)'s region ends
9: until CAS(\&\( W_x \), c@t, epoch(T))
10: if shouldRecord then
11: \( T.K[t] \leftarrow c \)
12: \( T.log.record(DEP, \text{siteID}, T.\text{ypc}, c@t) \)
13: \( x \leftarrow v \) \( \triangleright \) Program write
**Algorithm 3**  
**READ [RECORD]:** thread $T$ reads variable $x$

1: let $c@t ← W_x$
2: let conflict $←$ false
3: if $T.K[t] < c ∧ c = \text{clock}(t)$ then \hfill \triangleright t's region is ongoing
4: \hspace{1em} conflict $←$ true
5: \hspace{1em} $v ← x$ \hfill \triangleright Program read returns value $v$
6: if conflict $∨ W_x \neq c@t$ then \hfill \triangleright Reload $W_x$ for validation
7: \hspace{1em} $T$.log.record(VALUE, siteID, T.ypc, $v$) \hfill \triangleright Record value
8: \hspace{1em} else if $T.K[t] < c$ then \hfill \triangleright This dependence is not implied
9: \hspace{1em} $T.K[t] ← c$
10: \hspace{1em} $T$.log.record(DEP, siteID, T.ypc, $c@t$)
11: \hspace{1em} return $v$

**Algorithm 4**  
**COORDINATE(t, c) [RECORD]:** thread $T$ is the caller

1: repeat
2: \hspace{1em} let req $← t$.req
3: \hspace{1em} until CAS($&t$.req, req, req + 1) \hfill \triangleright T sends a request to $t$
4: repeat
5: \hspace{1em} let $c' ← \text{clock}(t)$
6: \hspace{1em} until $c' \neq c$ \hfill \triangleright T waits for $t$'s region ends

**Algorithm 5**  
**SAFE POINT [RECORD]:** thread $T$ at a safe point

1: let req $← T$.req
2: if req $> T$.reqSeen then
3: \hspace{1em} $T$.reqSeen $←$ req
4: \hspace{1em} region_end()
5: \hspace{1em} $T$.log.record(SAFEPOINT, siteID, T.ypc)
Atomicity. Algorithms 2 and 3 both preserve the atomicity of the instrumentation and the corresponding program access without using locks or complicated synchronization. This correctness guarantee is a combined result of the order of the instrumentation and program access, and invariants of the record algorithm. One important invariant is that a thread \( T \) only updates \( T.\text{K}[t] \) to \( c \) when the region \( \text{c@t} \) has ended. This property is ensured by always checking if the region \( \text{c@t} \) is ongoing, before updating \( T.\text{K}[t] \).

Algorithm 2 ensures atomicity from when line 9 successfully changes \( W_x \), to the end of the current region, including the program write on line 13. If two threads \( t_1 \) and \( t_2 \) try to write to \( x \) concurrently for the first time in their regions, only one of them (say \( t_1 \)) can succeed on line 9. \( t_2 \) would loop and reload the metadata \( W_x \), which could be of the value \( \text{c@t1} \). If the region \( \text{c@t1} \) is still ongoing, the condition check \( t_2.\text{K}[t1] < c \) on line 5 must be true for \( t_2 \), so \( t_2 \) will coordinate with \( t_1 \) and wait till \( t_1 \)’s region ends, preserving atomicity.

Observing that only Algorithm 2 modifies the metadata \( W_x \), the recording algorithm can be reasoned as the following. A variable \( x \) is implicitly “locked” by an ongoing region \( \text{c@t} \) when the first write access changes \( W_x \) to \( \text{c@t} \). When the region ends, it “unlocks” all variables it has written to. A concurrent write to \( x \) would request and wait until the region \( \text{c@t} \) ends via coordination, preserving the atomicity around instrumentation and the access. The clock values in \( \text{K} \) helps determine if a variable is definitely “unlocked”.

In order to ensure atomicity of Algorithm 3, for a read access to variable \( x \) that is not “locked”, the instrumentation speculatively loads the metadata \( W_x \) before and after the program read, validating if both loads return the same value. Since Algorithm 2 stores each epoch value \( \text{c@t} \) to \( W_x \) at most once, the validation does not have the ABA problem. If the validation fails or \( x \) is “locked” in the first place, Algorithm 3 just records the value.
of the read, which is independent from the value of $W_x$ and does not need the atomicity requirement.

### 4.2.4 Replay

Algorithms 6 to 9 describe the replay algorithm. At a high level, the replay algorithm explicitly enforces the recorded write–write and write–read dependences by either making the sink of a dependence edge wait until the source has executed, or returning a recorded value. The algorithm checks if any read–write dependence is violated (flipped), and recovers the correct value for the read if so.

A critical observation in RegPlay’s replay algorithm is that every flipped read–write dependence can be detected at run-time, even if no information is recorded for the read access (due to transitive relations). This valuable property is the result of preserving two important invariants between the recorded and replayed executions: (a) all writes to a variable $x$ maintain the same total order; (b) any update to a thread’s dependence vector clock $K$ occurs at the same dynamic program location with the same value.

When a flipped read–write dependence is detected, there could be several ways to handle the replay execution. The replay execution could just fail the execution and retry from the beginning, hoping the read–write dependence not getting flipped next time. Alternatively, the replay execution could record the original read–write dependence, effectively patching recorded dependences, then retry the replay execution from a previous checkpoint while enforcing the additional read–write dependence. However, both approaches are unsatisfactory because they do not guarantee a successful replay execution. A later replay execution could detect a different flipped read–write dependence and then fails or restarts.
Instead, with the help of the two above-mentioned invariants, RegPlay’s replay algorithm uses an approach that recovers the value for the read in a flipped read–write dependence on the fly, ensuring success and determinism of replay executions. The replay algorithm maintains the following data structure for each shared-memory variable to support recovering the value of the read:

\( M_x \) : A write log that keeps information about previous writes to variable \( x \). \( M_x \) is an ordered list of \( \langle v, c@t \rangle \) tuples. Each tuple denotes that thread \( t \) writes to \( x \) at clock \( c \) with value \( v \). The order of the entries in the list corresponds to the order that the writes happened to \( x \).

Note that maintaining \( M_x \) is only required for recovering from flipped read–write dependences, but not for detecting them. The replay algorithm relies on the dependence vector clock to detect flipped dependences.

Algorithm 6 shows the instrumentation when thread \( T \) writes variable \( x \) with value \( v \). Line 1 first checks if the current dynamic program location matches the next event to be replayed. If the check succeeds, the next event is guaranteed to be of the type \( \text{DEP} \). The instrumentation waits for the recorded region \( c@t \) ends (lines 3–4), ensuring the order of the write–write dependence. Then \( T \) updates its dependence vector clock with \( c@t \), just as during record. Line 6 finishes replaying the current \( \text{DEP} \) event and reads the next recorded event. Finally, the instrumentation updates \( W_x \) with the current epoch, and append this write event to the write log \( M_x \). Ideally, \( W_x \) only needs to be updated at the first write to \( x \) in a region, but the first write may not be recorded. The algorithm simply updates \( W_x \) at every write, since loading \( W_x \) and checking if it is the same as \( \text{epoch}(T) \) is probably more expensive. In addition, modifications to \( W_x \) and \( M_x \) do not need any synchronization, because the order of writes to \( x \) is strictly enforced.
Algorithm 7 shows the instrumentation that replaces a program read that a thread $T$ reads variable $x$. The original program read is at line 9. The instrumentation could return a value different from the one loaded from $x$, as a result of line 3 or 12. Lines 1 to 8 are similar to the instrumentation before a write access (Algorithm 6), except that lines 2 and 3 check and possibly return a recorded value if the program read was involved in a region conflict during record. When the program read executes at line 9, the replay algorithm guarantees that the write access that this read should read from has already executed. The condition on line 11 detects flipped read–write dependences after the program read. If $T$ has already observed the last writer region from $W_x$, the read must return the same value as during record. Otherwise, line 7 of Algorithm 6 for a write that occurs after the read during record has executed before line 10 loads $W_x$, and a flipped read–write dependence has possibly occurred. In the latter case, $T$ searches the write log $M_x$ for the correct value to return (line 12), using Algorithm 8. Section 4.2.6 proves that Algorithms 7 and 8 are guaranteed to return the correct value for the read.

Algorithm 6

```plaintext
WRITE [REPLAY]: thread $T$ writes variable $x$ with value $v$

1: if $T$.ypc = $T$.log.nextYpc ∧ siteID = $T$.log.nextSiteID then
2:   let c@t ← $T$.log.nextEpoch
3:   while c <= clock(t) do ▷ Wait till recorded region ends
4:     spin()
5:   T.K[t] ← c
6:   T.log.readNextEvent()
7:   $W_x$ ← epoch(T)
8:   $M_x$.append($v$, epoch(T)) ▷ Program write
9:   $x$ ← $v$
```

Algorithm 8 describes the search algorithm for recovering the correct value for a read when a flipped read–write dependence occurs. The algorithm iteratively checks each entry
**Algorithm 7**

**READ [REPLAY]:** thread \( T \) reads variable \( x \)

1. if \( T\.ypc = T\.log\.nextYpc \land siteID = T\.log\.nextSiteID \) then
2.     if \( T\.log\.type = VALUE \) then
3.         return \( T\.log\.value \)
4.     let \( c@t \leftarrow T\.log\.nextEpoch \)
5.     while \( c \leq \) clock(t) do \( \triangleright \) Wait till recorded region ends
6.         spin()
7.     \( T\.K[t] \leftarrow c \)
8.     \( T\.log\.readNextEvent() \)
9.     \( v \leftarrow x \) \( \triangleright \) Program read
10. let \( c'@t' \leftarrow W_x \)
11. if \( T\.K[t'] < c' \) then \( \triangleright \) Flipped write access detected
12. \( v \leftarrow M_x\.search() \) \( \triangleright \) Recover the correct value
13. return \( v \)

---

**Algorithm 8**

**SEARCH [REPLAY]:** thread \( T \) searches \( M_x \)

1. let \( i \leftarrow M_x\.size() \)
2. repeat
3.     \( i \leftarrow i - 1 \)
4.     let \( \langle v, c@t \rangle \leftarrow M_x\.get(i) \)
5. until \( T\.K[t] \geq c \)
6. return \( v \)
in $M_x$ in reverse order, until it finds an entry with an epoch that $T$ has already seen. The value in this entry must be the correct value for the read to return.

Algorithm 9

SAFE POINT [REPLAY]: thread $T$ at a safe point

1: if $T$.ypc = T.log.nextYpc ∧ siteID = T.log.nextSiteID then
2: region_end()
3: T.log.readNextEvent()

Algorithm 9 shows the instrumentation at a safe point. The instrumentation checks if the dynamic program location matches the next event to replay. If so, $T$ ends its current region and loads the next event to replay.

The write log $M_x$ can be optimized based on its limited use cases. In fact, only the last write to $x$ in a region needs to be stored in $M_x$. This is because Algorithm 8 can only return the value of the last write in a region. Thus, instead of appending an entry at every write in Algorithm 6, a write to $x$ could check if it is the first write to $x$ in the region. If so, $T$ loads the values from $x$ and $W_x$, and appends them to $M_x$. Then $T$ proceeds to change $W_x$ and write to $x$.

Another observation is that even Algorithm 8 could load from $M_x$ concurrently with Algorithm 6 storing to $M_x$, it is possible to avoid expensive synchronization on $M_x$ in both algorithms. The reason is that when Algorithm 8 executes, it is guaranteed to be able to find an entry in $M_x$ satisfying the condition on line 5, because the write that the read should read from must have already executed.

4.2.5 Insights and Rationale

RegPlay employs the following insights in its design, in order to achieve low recording overhead and preserve replay determinism.
Sources for dependence edges. RegPlay records a cross-thread dependence as a happens-before edge [77] that starts from a region boundary and ends at the access, which transitively implies the dependence. The source of the recorded edge does not precisely identify the location of the instruction involved in the cross-thread dependence.

The rationale behind this design choice is based on the characteristics of DRF accesses. For any cross-thread dependence edge that does not constitute a data race, the two accesses must have established a happens-before relation. The source of the happens-before relation must be a program release operation executed after the first access by the first thread. Since every program release operation is a region boundary, RegPlay can always record such an edge from a region boundary to the second access for a DRF dependence. As dynamic instances of data races are rare in a program execution [23, 93], most dependences can be recorded in this manner.\footnote{More precisely, only data races that also constitute region conflicts would require an alternative way to record the dependences.}

This design makes it unnecessary to store the precise location of the last access—the epoch information suffices. Moreover, multiple accesses to same variable in the same region share the same epoch as the location, eliminating the need to store a different location at each access. Many existing approaches that explicitly record dependences actively maintain a precise location information for the last access, such as a per-thread instruction counter. Storing such precise location information at every access is expensive. We find that our design for sources of dependences, together with the insight about transitively implied dependences, helps avoid recording many redundant dependence edges.

Avoiding maintaining last reader information. It is unnecessary and expensive to maintain information about the last reader (thread or instruction) after every read access. The...
last reader information is required for detecting and recording read–write dependences. Nonetheless, it is possible to provide replay determinism without recording read–write dependences. RegPlay shows that as long as the record and replay algorithms soundly preserve all write–read and write–write dependences, the replay algorithm can detect any violation of read–write dependences and recover the original value for the read. Our experience shows that violations of read–write dependences are infrequently during replay, unless the recorded execution has many dynamic instances of data races.

Maintaining last reader information typically incurs a store at every read access, which turns a read-only access into a write access, significantly degrading cache performance. Since read accesses are more frequent than write accesses, such overhead at a read access drastically slows down overall execution. This insight echoes with insights from previous research, which also eliminates stores to shared-memory at read accesses [16, 85, 89, 123].

**Transitively implied dependences.** Many cross-thread dependences are transitively implied by other dependences. As the example in Figure 4.1 shows, it is sufficient to only record the “shortest” dependences (i.e., dependences that are not implied by others). Some existing record & replay approaches can avoid recording transitively implied dependences involving only one variable, but few tries to optimize for the general case of transitively implied dependences. This is probably due to the sheer number of cross-thread dependences involved in an execution, so that maintaining an accurate partial order of dependences among threads is more expensive than just recording the dependences.

RegPlay uses epoch as the source of a dependence, making it convenient to maintain the dependence vector clock (Section 4.2.3) at every access. The dependence vector clock effectively tracks the order established by cross-thread dependences between any two
Figure 4.4: An example transitive dependence involving three threads. The write–read dependence on x between T1 and T3 is transitively implied by the dependences between T1 and T2 on x, and between T2 and T3 on y.

threads, similar to the traditional vector clock algorithm maintains the partial order of program synchronization operations [97]. Thus, RegPlay is able to avoid recording many transitively implied dependences between any two threads.

It’s worth mentioning that the dependence vector clock can not track transitive dependences involving three or more threads. Figure 4.4 show such an example. In this example, suppose all dependences are not involved in region conflicts. The write–read dependence on x between T1 and T3 is transitively implied by the other two write–read dependences. However, when T3 performs rd x, its dependence vector clock has not been updated to reflect that T1’s region for wr x has ended. In this case, T3 would explicitly record the dependence on x. Nevertheless, recording and replaying this dependence does not affect correctness.

It would be significantly more expensive to soundly track the partial order for all dependences, as it would require storing a whole vector clock instead of just the epoch at every access, including read accesses. Our design of dependence vector clock trades soundness of tracking all partial order for performance. The dependence vector clock neither tracks the partial order of dependences among all threads, nor the partial order established by read–write dependences. The condition that a write access of thread T1 is ordered before
another thread T2 according to T2’s dependence vector clock, is equivalent to the fact that some write–read or write–write dependence from a write in T1 to T2 has occurred.

4.2.6 Correctness of Algorithm 8

This section proves that Algorithm 8 always provides the correct value for a read when Algorithm 7 detects a flipped read–write dependence in replay. The value for the read is correct if and only if the read returns the value written by the same write access as it occurs during record. This section first reasons about two important arguments about the replay algorithm, then analyzes the structure of the write log $M_x$ when a flipped read–write dependence is detected, which explains the correctness of Algorithm 8.

**Theorem 1.** Algorithm 7 ensures that when a program read executes during replay (line 9 in Algorithm 7), the write that the read should read from has already executed.

**Proof.** If the write is by the same thread, the proof is trivial since record and replay maintain the same program order. Thus we only consider the case that the write is from a different thread.

For a read whose dynamic program location is recorded, Algorithm 7 ensures that the region containing the desired write has already ended, by the explicit waiting on lines 5–6. Thus the write must have already executed.

For a read $r$ whose dynamic program location is not recorded, suppose $r$ reads from the write $w$. The record algorithm has determined that the write–read dependence is transitively implied by other dependences because $T.K[t] >= c$ on line 8 of Algorithm 3. The access $a$ by $T$ that updates $T.K[t]$ to make it satisfy this condition must have been recorded, and $a$ must execute before $r$, and after the region containing the $w$ has ended. During replay,
a must have been successfully replayed when T executes r, which ensures that the region containing w has ended. Thus, w must have already executed when r executes.

\[ \square \]

**Theorem 2.** When replaying a read that did not trigger a region conflict in record, \( T.K[t] < c \) is a necessary and sufficient condition for a flipped read–write dependence, where \( c@t \) is the write’s epoch.

*Proof.* A flipped read–write dependence in replay means that the write executes after the read in record, but the same write executes before the read in replay. Because replay ensures that each thread’s dependence vector clock \( K \) and each variable’s metadata \( W_x \) are updated in the same way as in record, it is equivalent to show that \( T.K[t] < c \) is a necessary and sufficient condition for the write to execute after the read in record.

**Necessity.** If the write executes after the read in record, the write’s region \( c@t \) must have not ended when the read executes. Since thread T only updates \( T.K[t] \) with c after the region \( c@t \) has ended, \( T.K[t] \) must contain a clock that is smaller than c.

**Sufficiency.** It is equivalent to show that if the write executes before the read in record, \( T.K[t] >= c \) must hold when the read executes. Since the read does not trigger a region conflict, the write’s region \( c@t \) must have ended when the read executes. Because at the read Algorithm 3 actively checks the condition \( T.K[t] < c \) (line 8) and updates \( T.K[t] \) with c if the check fails, \( T.K[t] >= c \) must hold at the read.

\[ \square \]

**Theorem 3.** At the moment when a read detects a flipped read–write dependence in replay, the write log \( M_x \) must contain a write \( w_e \) that occurs before the read during record. Besides, all writes before \( w_e \) in \( M_x \) have occurred before the read in record, and all writes after \( w_e \) in \( M_x \) have occurred after the read in record (i.e., they are all flipped to the read).
Proof. Theorem 1 shows that the write that the read should read from must have already executed. \( M_x \) must contain this write, and this write must occur before the read during record. Thus, \( w_e \) must exist.

Since \( M_x \) maintains the total order of writes exactly the same as during record, suppose \( M_x \) also exists during record. At the moment when the read executes in record, all writes in \( M_x \) occur before the read, and the read must return the value of the last write \( w_l \). All writes execute after the read during record are candidates for flipped writes to the read during replay, and in \( M_x \) these writes would all be after \( w_l \). The scenario when a read detects a flipped write in replay is equivalent to the case that the read executes later than it executed during record. At this moment in replay, \( M_x \) would have the same content as the supposedly-existing \( M_x \) in record at a write that executes after the read. Thus, \( w_l \) satisfies the condition for \( w_e \) described in the theorem.

The proof for Theorem 3 also shows that \( w_e \) is the write that the read should read from. According to Theorem 2, the condition \( T.K[t] < c \) soundly identifies whether a write should occur before the read. Therefore, Algorithm 8 would always find the correct write whose value should be returned for the read.

4.2.7 Alternative Approach to Recover Flipped Dependences

As described at the beginning of Section 4.2.4, it is possible to use an alternative approach to handle flipped read–write dependences. This section introduces an approach called iterative patching, that recovers flipped read–write dependences using an iterative, trial-and-error methodology.

Maintaining the write log during replay could be expensive, as it would take up large memory space if the program frequently writes to memory. For programs with few data
races or flipped read–write dependences, the cost of maintaining the write log may not justify the benefits and guarantees it provides. Iterative patching is a good alternative in such situations, as well as when the system is already using certain checkpointing or speculation mechanisms that supports rollback and retry.

At a high level, when a replay execution detects a flipped read–write dependence, the iterative patching approach records this read–write dependence, which essentially patches initially recorded dependences with a new dependence. Then the approach retries the replay execution from a previous checkpoint (or the beginning of the execution), enforcing the newly recorded read–write dependence as well as initially recorded dependences. This process repeats until the replay execution no longer detects a flipped read–write dependence and successfully terminates. The execution is guaranteed to be replayed successfully eventually, because there are finite number of read–write dependences.

There are two main challenges for the iterative patching approach. First, when a read detects a flipped read–write dependence in replay, the read is not aware of the dynamic program location of the write. This is because the replay algorithm only maintains the epoch, but not the ypc or siteID of the last write. Our solution is to conservatively treat the first write to any variable in the epoch as the write in the flipped read–write dependence.

Second, flipped read–write dependences could be detected out of program order among multiple replay attempts. For example, in the first replay attempt thread t1 detects a flipped dependence at read r. In the second replay attempt, t2 could detect another flipped dependence, whose write is by t1 in an epoch that precedes r. Our solution to this problem is to maintain separate log files for read and write events, and support sorting the write events.

Below is a detailed description for the iterative patching approach:
1. Across all replay executions, each thread maintains two additional log files for storing read–write dependences: one for events at reads (rLog), one for events at writes (wLog). This design avoids maintaining the order between read and write events in one log file. However, the approach still maintains the order among read events, and the order among write events.

2. When thread \( t_1 \) detects a flipped dependence at a read whose dynamic program location is \(<\text{ypc}_1, \text{sitelD}_1>\), with the write’s epoch being \( c@t_2 \), \( t_1 \) immediately suspends all other threads, and records the following events. In \( t_1.\text{rLog} \) it records \(<\text{ypc}_1, \text{sitelD}_1>, \) only if \( t_1.\text{rLog} \) does not already contain this dynamic program location. In \( t_2.\text{wLog} \), \( t_1 \) records the following tuple:

\(<\text{GC\_count, c, t1, ypc1, sitelD1}>\)

3. \( t_1 \) would record the read event before the next read event that \( t_1 \) is about to replay in \( t_1.\text{rLog} \). \( t_1 \) inserts the write event at an appropriate location in \( t_2.\text{wLog} \) according to the value of \( \text{GC\_count} \) and \( c \), so that events in \( t_2.\text{wLog} \) remain sorted in program order. Then \( t_1 \) retries this replay execution from a previous check point.

4. In the next replay attempt, each thread maintains a set of sitelDs (readSites) for read events with the same ypc that are about to be replayed. At a read, the approach first runs Algorithm 7 without line 12. If the read detects a flipped dependence, it repeats steps 2 and 3 and terminates. Otherwise, after the read, if the current ypc matches the next replay ypc for read events, and the current sitelD is in readSites, this thread would remove the sitelD from readSites. If readSites becomes empty, the thread loads read events to be replayed with the same ypc.
5. At the first write in a region, if the current `GC_count` and the region’s clock matches the next event in `wLog` to be replayed, the thread waits for the reader thread’s current `ypc` to exceed the recorded `ypc`, or when the reader thread’s `readSites` no longer contains the recorded `siteID`.

A limitation for the iterative patching algorithm is that replay could deadlock when enforcing two read–write dependences in some subtle cases. Figure 4.5 shows one such example. The figure shows the interleavings and dependences during record. Initially, no dependence between T1 and T2 is recorded. Suppose the first replay attempt detects a flipped read–write dependence on x, so it patches the logs to make T2’s `wr x` wait for T1’s `rd x` to finish. In the second replay attempt, suppose T2 detects a flipped dependence on y, so it tries to make T1 wait for T2’s `rd y` to finish at the `write to z`, which is the first write in T1’s region. In the third replay attempt, T1’s `wr z` waits for T2’s `rd y`, and T2’s `wr x` waits for T1’s `rd x`, forming a deadlock.

The root cause for this problem is that the approach does not record a precise dynamic program location of the write in a read–write dependence. When a thread tries to enforce a read–write dependence and waits at the first write to any variable in the region, the thread could be waiting at a point that is “too early”, which precedes a read of another read–write dependence that should be replayed before waiting. It is possible to modify the iterative patching approach to avoid such deadlocks. For example, a read in a read–write dependence could record the returned value when a write in another read–write dependence happens in the same region as the read.

Finally, as Theorem 3 describes, when a read detects a flipped read–write dependence, it is possible the several writes to the object or static field that occurred after the read in record have executed before the read (i.e., multiple read–write dependences involving the same
read are flipped). In this case, the iterative patching approach only tries to fix one read–write dependence at a time. It could take several replay attempts to fix all read–write dependences involving the same read.

4.3 Implementation

We have implemented RegPlay with the iterative patching approach in Jikes RVM 3.1.3 [8, 9], a high-performance Java virtual machine (JVM) that performs competitively with commercial JVMs [16]. Our implementation reuses part of publicly available implementation of an existing multithreaded record & replay in Jikes RVM [24], in order to deal with challenges that hinders deterministic execution in a runtime for a managed language.

**Application-level determinism.** Similar to prior work, the goal of RegPlay’s implementation is to provide *application-level determinism*: the record and replay executions should appear identical from the application’s perspective, but the JVM internally does not need to execute deterministically [24]. Prior work has addressed many challenges for providing application-level determinism in Jikes RVM. For example, the existing implementation
records and replays garbage collection events, supports deterministic hash code, and pro-
vides deterministic results for I/O and some system calls. Our implementation reuses these
features and mechanisms in order to demonstrate RegPlay’s replay algorithm.

Two features from prior work are crucial to understand our implementation and exper-
iments. (1) There are two versions of each library method: one is of application context
and called by application code, the other is of VM context and used by VM. The imple-
mentation only instruments application-context library methods. (2) In order to deal with
challenges from dynamic class loading which tightly couples VM and application activities,
the implementation uses a research methodology called fork-and-recompile. The JVM
executes two iterations of the program. The first iteration warms up the program, then the
JVM forks a child process to run the record execution. The parent process waits for the
child process to finish, then runs the replay execution. Both record and replay executions
first recompile all application-context methods with instrumentation for record and replay
algorithms, respectively.

**Object granularity.** In order to incur low overhead for the record algorithm, the imple-
mentation tracks dependences at object level for shared-memory variables in the heap. Each
object (scalar or array) maintains one metadata word for the epoch of last writer region ($W_x$,
Section 4.2.2). Each static field adds one metadata word in the global table of statics for the
same purpose. Our implementation targets the IA-32 platform, so words are 32 bits. An
epoch $\text{c@t}$ is encoded in one word using the clock value and the thread’s ID.

Tracking dependences at object level over-approximates the number of cross-thread
dependences, but still soundly captures all real dependences. Object granularity does not
affect the correctness of our record and replay algorithms, although it may add complexity
to the implementation of the write log ($M_x$, Section 4.2.4) used in the replay algorithm.

**Handling flipped read–write dependences.** Our prototype implementation encountered
a challenge in efficient implementing the write log ($M_x$, Section 4.2.4) for the replay
algorithm. The write log requires field granularity in order to correctly support Algorithm 8,
but the record implementation tracks dependences and adds metadata at object granularity.
Although in general it is possible to maintain addition per-field metadata only during replay,
it is difficult to do so using the fork-and-recompile methodology. In Jikes RVM, it is hard
to change field layout of objects dynamically after a program has finished the warmup
iteration. 12

Another assumption for our implementation is that flipped read–write dependences
should be rare in practice, due to ordering constraints of write–write, write–read dependences
and synchronization operations. As a result, instead of maintaining the write log, our
implementation uses the iterative patching approach (Section 4.2.7) to handle flipped read–
write dependences.

Our implementation does not support checkpoint. The iterative patching approach
restarts a replay execution from the beginning. In our experiments, we do not encounter the
deadlock described in Section 4.2.7. Thus, our implementation does not employ mechanisms
to avoid the deadlock.

**Instrumentation.** The implementation modifies Jikes RVM’s baseline and optimizing
compilers to insert instrumentation in all application and application-context library methods.

12 Recently we have a new idea to sidestep this challenge, which adds a level of indirection for a field to
locate its write log. We haven’t implemented this idea yet.
The implementation adds barriers at every read and write to an object field, static field, or array element, except for accesses to a few objects and fields that are immutable, such as String objects and final fields. The actual code for read and write barriers is semantically equivalent to the algorithms presented in Sections 4.2.3 and 4.2.4. However, the implementation has separated a barrier into a fast-path check that is inlined to the application code by the optimizing compiler, and a non-inlined slow-path method that only executes when the fast-path check fails. In order to deal with different types of values efficiently, the implementation uses macros to generate customized read barriers for each value type.

The implementation also instruments every program synchronization release operation such as lock release, monitor wait and thread fork to demarcate regions and maintain the clock (Section 4.2.2). In order to prevent the clock from overflowing, we have modified the garbage collector to reset the clock and dependence vector clock of each thread, and the epoch metadata of each object and static field at every full-heap garbage collection. This is because it is unnecessary to record and replay a dependence that crosses a full-heap garbage collection in our implementation. Since Jikes RVM by default uses a stop-the-world garbage collector, and the implementation already records and replays each garbage collection event, a full-heap garbage collection serves as a global synchronization barrier that is enforced at the same dynamic program location of every thread during both executions.

Finally, the implementation adds instrumentation to every method entry, method return, loop back edge, and potential blocking operation to maintain the yield point counter (Section 4.2.2) and respond to coordination (Algorithms 5, 9). The implementation also instruments every program synchronization operation in order to record and replay them. In order to improve responsiveness of the coordination and prevent deadlocks, the implementation also supports implicit coordination. An important note is that in Algorithm 4,
lines 4 to 6 are treated as a blocking operation and surrounded by safe point instrumentation for implicit coordination. This safe point ensures deadlock freedom of the coordination mechanism during record.

**Recording and replaying with reference values.** Algorithm 3 requires recording the value of a program read if the read triggers a region conflict. In the replay execution, Algorithm 7 returns the recorded value directly for this program read. However, recording and replaying values of reference types poses a problem. Since the implementation only provides application-level determinism, the addresses of objects are not generally the same across record and replay executions.

The implementation uses a workaround to this problem. During record, the instrumentation records the hash code of the object. During replay, the instrumentation checks if the object from the program read has the same hash code. If so, replay deems that the object is the same as during record and continues execution. Otherwise, replay waits with a timeout until the read returns an object with the recorded hash code. If the timeout expires, the replay fails the execution with an error.

The implementation uses this simple workaround instead of directly addressing the problem because it is rare for a reference-type read to trigger a region conflict. Even such a read occurs, in replay the same read is probably able to return the same object immediately or after a little wait, due to enforcement of other dependences and synchronization ordering.

Note that this problem would not exist if the implementation targets programs in native languages such as C or C++, or if the implementation makes the runtime fully deterministic. It is also possible to solve this problem in the current implementation using the following approach. Each object can be uniquely tagged by the dynamic program location of its
allocation site. During record, the read would record the tag as well as the type of the object. Replay maintains a map from a tag to the address of corresponding object for each recorded type. The read can thus look up for the object’s address based on recorded tag and type.

### 4.4 Evaluation

This section evaluates the performance and effectiveness of RegPlay, comparing with an existing multithreaded record & replay approach [24], which this section refers to as “Roctet”.

#### 4.4.1 Methodology

The experiments execute benchmarks consisting of large, parallel, real-world applications: the DaCapo benchmarks [17] versions 2006-10-MR2 and bach-9.12 (distinguished with suffixes 6 and 9), fixed-workload versions of SPECjbb2000 and SPECjbb2005. For the DaCapo benchmarks, we exclude programs that are single-threaded, or that unmodified Jikes RVM cannot reliably execute, or that do not work with the fork-and-recompile methodology. The experiments also exclude pmd9 because it utilizes synchronization idioms from java.util.concurrent package that our implementation does not currently track.

We build a high-performance configuration of Jikes RVM (FastAdaptiveImmix) that adaptively optimizes compiled code at run time and adjusts the heap size automatically. The experiments use a non-generational stop-the-world garbage collector (Immix), so that full-heap garbage collection is triggered more often, which helps avoid overflowing the per-thread clock $c$ (Section 4.3). If the implementation targets at a 64-bit platform, the
per-thread clock would probably rarely overflow, so experiments could just use a more efficient generational garbage collector.

The experiments run on a 32-core system with four 8-core Intel Xeon E5-4620 processors running Linux 2.6.32. To account for run-to-run variability, each performance result is the median of 10 trials, and each statistic result is the mean of 5 trials. All experiments use the fork-and-recompile methodology. We have modified the fork-and-recompile mechanism to support forking multiple replay processes, facilitating the iterative patching approach.

4.4.2 Runtime Characteristics

4.4.2.1 Record

Table 4.1 shows runtime statistics of recorded executions for Roctet and RegPlay. The first two columns show the numbers of total and maximum live threads for each program. 9 out of the 10 programs spawn more than 8 threads, which indicates the degree of parallelism that the analyses need to deal with.

The three columns under “Roctet” show the numbers of recorded dependences and the average size of log files for a single execution. The “Total” column counts all recorded dependences. The column “Excl. rd–wr” is a subset of “Total”, which only counts write–write and write–read dependences. In most cases, when Roctet records a dependence, it triggers heavyweight inter-thread communication, which is quite expensive as prior work has shown [24, 25, 34]. The numbers of recorded dependences (and thus the size of log files) vary significantly across programs. It is generally true for Roctet that the more dependences recorded, the higher the recording overhead (Section 4.4.3).

The remaining five columns show the numbers of recorded dependences, log file sizes and number of dynamic regions for RegPlay. The columns “Coordination” and “Value” are subsets of the “Total” column, which count write–write and write–read dependences that
<table>
<thead>
<tr>
<th>Threads</th>
<th>Roctet</th>
<th>Log (MB)</th>
<th>RegPlay</th>
<th>Log (MB)</th>
<th>Regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Live</td>
<td>Total (Excl. rd–wr)</td>
<td></td>
<td>Total</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hsqldb6</td>
<td>402 102</td>
<td>9.9×10^5</td>
<td>(9.2×10^5)</td>
<td>190</td>
<td>2.9×10^5</td>
</tr>
<tr>
<td>lusearch6</td>
<td>65 65</td>
<td>6.4×10^3</td>
<td>(6.3×10^3)</td>
<td>7.3</td>
<td>3.3×10^2</td>
</tr>
<tr>
<td>xalan6</td>
<td>9 9</td>
<td>1.8×10^7</td>
<td>(1.8×10^7)</td>
<td>365</td>
<td>4.6×10^6</td>
</tr>
<tr>
<td>avrora9</td>
<td>27 27</td>
<td>1.0×10^7</td>
<td>(9.7×10^6)</td>
<td>430</td>
<td>8.1×10^6</td>
</tr>
<tr>
<td>luindex9</td>
<td>2 2</td>
<td>2.4×10^2</td>
<td>(2.3×10^2)</td>
<td>7.5</td>
<td>3.0×10^0</td>
</tr>
<tr>
<td>lusearch9</td>
<td>32 32</td>
<td>2.7×10^3</td>
<td>(2.5×10^3)</td>
<td>7.9</td>
<td>6.8×10^2</td>
</tr>
<tr>
<td>sunflow9</td>
<td>64 32</td>
<td>3.3×10^4</td>
<td>(3.2×10^4)</td>
<td>8.3</td>
<td>1.4×10^4</td>
</tr>
<tr>
<td>xalan9</td>
<td>32 32</td>
<td>1.9×10^7</td>
<td>(1.9×10^7)</td>
<td>352</td>
<td>6.8×10^6</td>
</tr>
<tr>
<td>pjobb2000</td>
<td>37 9</td>
<td>1.5×10^6</td>
<td>(1.5×10^6)</td>
<td>31</td>
<td>2.3×10^4</td>
</tr>
<tr>
<td>pjobb2005</td>
<td>9 9</td>
<td>6.1×10^7</td>
<td>(5.2×10^7)</td>
<td>2078</td>
<td>4.4×10^7</td>
</tr>
</tbody>
</table>

Table 4.1: Characteristics of recorded executions for Roctet and RegPlay. The columns named “Total” show the total numbers of recorded dependences for each analysis.
trigger region conflicts during record, respectively. Dependences that cause region conflicts are rare for most programs, except for avrora9 and pjbb2005, which are actually due to frequent true data races (Section 4.4.2.2).

The difference between the “Total” columns of Roctet and RegPlay shows how many dependences that RegPlay avoids to record in total. The difference between the “Excl. rd–wr” column and RegPlay’s “Total” column illustrates how many transitively implied dependences that RegPlay skips thanks to its use of the dependence vector clock (Section 4.2.2). RegPlay manages to record dependences that are an order of magnitude fewer than Roctet records, while still preserving replay determinism.

The savings in log file sizes for RegPlay does not completely correlate with RegPlay’s fewer recorded dependences than Roctet’s. This is because RegPlay records program synchronization operations, while Roctet does not. RegPlay records synchronizations in order to reduce the number of flipped read–write dependences in replay, which helps the iterative patching process finish sooner. However, it is possible not to record synchronizations in order to trade for lower space (and likely record) overhead.

Currently the implementation tracks synchronization through an approach that is independent of RegPlay’s record algorithm. The order established by program synchronizations is not reflected in the dependence vector clock. It is possible to instrument a synchronization acquire operation in the same way as for a program write, which could avoid recording many cross-thread dependences and synchronizations as they could be transitively implied by each other.

4.4.2.2 Replay

Our experiments use the iterative patching approach (Section 4.2.7) to recover from a flipped read–write dependence. It might take multiple replay attempts until an execution is
Attempts

<table>
<thead>
<tr>
<th>Program</th>
<th>Attempts</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb6</td>
<td>10</td>
</tr>
<tr>
<td>lusearch6</td>
<td>1</td>
</tr>
<tr>
<td>xalan6</td>
<td>1</td>
</tr>
<tr>
<td>avrora9</td>
<td>74*</td>
</tr>
<tr>
<td>luindex9</td>
<td>1</td>
</tr>
<tr>
<td>lusearch9</td>
<td>119</td>
</tr>
<tr>
<td>sunflow9</td>
<td>68*</td>
</tr>
<tr>
<td>xalan9</td>
<td>1</td>
</tr>
<tr>
<td>pjbb2000</td>
<td>55*</td>
</tr>
<tr>
<td>pjbb2005</td>
<td>47*</td>
</tr>
</tbody>
</table>

Table 4.2: Average number of replay attempts until replay succeeds using the iterative patching approach. *These programs utilize extra techniques after profiling to reduce flipped read–write dependences.

successfully replayed. In order to validate a correctly replayed execution, when a thread terminates, the implementation checks if there are no remaining events to be replayed in this thread’s log. A replay attempt could also fail in the middle of execution: when a read detects a flipped read, or when a reference-type read that is recorded by hash code times out waiting to see an object with the same hash code. Nevertheless, our experiments never encounter a failed replay attempt due to the latter issue.

Table 4.2 shows the average number of replay attempts for each program. 4 out of 10 programs can usually succeed replaying the execution in the first attempt, demonstrating that all read–write dependences are likely already implied by other enforced ordering constraints. hsqldb6 and lusearch9 encountered a few flipped read–write dependences, but the iterative patching approach can fix them within a small number of replay attempts.

For the programs marked with asterisk, the iterative patching approach as described in Section 4.2.7 incurs many replay attempts (more than 1000) without success. After further investigation and profiling, we found that these flipped dependences are due to frequent
occurrences of object-level data races, which violates our assumption that they are rare in practice. We use two different techniques to help reduce the number of flipped dependences, so that replay can succeed within reasonable number of attempts.

For sunflow9 and pjbb2000, the object-level races are not true data races. Both programs have a similar access pattern for the problematic object. Most fields of the object are read-only after initialization (but are not declared final), and they are frequently read by multiple threads. However, one or two fields of the object are actively modified by multiple threads in a critical section. We argue that this design of object layout would cause false sharing, a performance pitfall that developers should try to avoid. Our solution is to avoid instrumenting accesses to the well-synchronized fields, since the implementation already tracks surrounding program synchronizations. This workaround is analogous to applying a static race detector to filter out instrumentation to DRF accesses.

For avrora9 and pjbb2005, the flipped dependences are due to true data races. We argue that such frequent dynamic data races indicate a serious bug in the program, which the developer should fix first before trying to run RegPlay. It is still possible to make RegPlay successfully replay these programs. Our solution is to identify the static sites of these racy accesses, and instrument them specially to record the total order of the accesses (e.g., treat all of these accesses as if they are writes).

4.4.3 Performance

Figure 4.6 shows the performance of record execution for RegPlay, compared with Roctet. These results do not use the techniques described in Section 4.4.2.2 for reducing flipped dependences. In addition, RegPlay tracks program synchronization operations, but Roctet does not. RegPlay outperforms Roctet for avrora9, xalan9, and pjbb2005, due to
recording less information and incurring fewer coordinations (Section 4.4.2.1). However, RegPlay is significantly more expensive than Roctet for hsqldb6, luindex9, sunflow9, and pjbb2000. On average, RegPlay’s record incurs 67% overhead, 16% (relative to baseline) more than Roctet’s 51% overhead. After further investigation, we found that RegPlay’s overhead mainly comes from its complex fast path for read barriers, which involves four loads and two conditionals. In contrast, Roctet’s read fast path is highly optimized, which only performs one load and one conditional. In summary, the complexity in RegPlay’s read fast path, together with the overhead for recording synchronization operations, offset most of the performance benefit from recording fewer dependences and incurring fewer coordinations than Roctet. As a future work, we will try to optimize RegPlay’s read fast path and experiment with configurations that do not track synchronizations.
We don’t show performance result for replay for two reasons. First, the iterative patching approach under the fork-and-recompile methodology makes it hard to measure replay performance. It does not make sense to compare the performance between iterative patching and Roctet, either. Second, we seek to implement RegPlay’s write log mechanism in the near future, which would make our implementation and evaluation more compelling.

4.5 Contributions and Impact

RegPlay is a novel, software-only multithreaded record & replay algorithm that provides replay determinism for all executions. RegPlay optimizes its record algorithm by logging only necessary write–read and write–write dependences in order to achieve low recording overhead, while its replay algorithm ensures replay determinism by checking and recovering read–write dependences. Experiments show that RegPlay only records a fraction of dependences comparing to existing approach, and achieves competitive performance. RegPlay advances the state of the art in multithreaded record & replay, overcoming limitations inherent to many existing approaches.

Contributions. This work makes the following contributions.

- We present a novel multithreaded record & replay algorithm that avoids recording many unnecessary dependences without compromising replay determinism. To our knowledge, RegPlay is the first approach that supports online replay for racy executions without recording read–write dependences. We prove that RegPlay’s replay algorithm soundly ensures replay determinism.
• RegPlay’s record algorithm effectively detects and skips many transitivity implied dependences involving multiple variables. The design of dependence vector clock could inspire and influence future analyses that requires efficient dependence tracking.

• Our implementation and evaluation demonstrate that RegPlay’s record algorithm adds low time and space overhead, while two distinct approaches could be used to achieve replay determinism.

Multithreaded record & replay has been studied for decades, but few approached are deployed in production systems. One major reason is that existing approaches have various limitations. RegPlay is able to handle racy executions effectively at low space and time overhead during record, without relying on custom hardware. RegPlay supports both offline and online replay, because it does not require an offline phase between record and replay. Existing approaches have struggled to provide these properties simultaneously. Thus, RegPlay helps the adoption of multithreaded record & replay system in production settings. RegPlay’s record algorithm can be incorporated into commercial VMs to provide always-on record support, suitable for monitoring and debugging production-time errors. RegPlay also enables designs that replicates running process to improve reliability, or that offloads other dynamic analyses to the replay process. These applications will have profound impact on improving the productivity of software development and debugging, as well as reliability and efficiency of production systems.
Chapter 5: Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future

This chapter presents a novel analysis called prescient memory, which exposes more program behaviors of real racy executions than existing approaches do, improving the coverage limitation in the state of art (Section 2.3). In particular, prescient memory exposes unknown program behaviors due to “future values” returned by loads involved in data races.

5.1 Problem and Motivation

As described in Chapters 1 and 2, a key problem with current shared-memory languages and hardware is that memory models allow many unexpected behaviors for executions containing data races. Researchers have introduced dynamic analyses that expose weak memory model behaviors [29, 54, 72, 74, 103, 124], but these approaches have limited coverage in the range of behaviors they can expose. Notably, most approaches cannot expose behaviors due to loading a “future value”—a value stored by a store that executes after the load that uses the value.

Figure 5.1 shows an example shared-memory program in a Java-like language; x and y are shared variables, and r1 and r2 are locals. Under a weak memory model such as Java’s memory model [92], it is legal for both loads to read the value 0, violating the assertion. However, such an outcome would not be possible if memory accesses appeared to execute
Initially $x = y = 0$

Thread 1:  
$y = 1;$  
$r1 = x;$

Thread 2:  
$x = 1;$  
$r2 = y;$

assert $r1 \neq 0 \;\|\; r2 \neq 0$

Figure 5.1: An assertion failure is possible under Java’s memory model. Existing dynamic analyses can expose the assertion failure.

Initially $x = y = 0$

Thread 1:  
$r1 = x;$  
$y = 1;$

Thread 2:  
$r2 = y;$  
$x = 1;$

assert $r1 == 0 \;\|\; r2 == 0$

Figure 5.2: An assertion failure is possible under Java’s memory model. Existing dynamic analyses cannot expose the failure.

in their original order (i.e., with sequential consistency (SC) semantics [78]). This kind of non-SC behavior not only is permitted in theory, but occurs in practice when compiler and hardware optimizations reorder intra-thread memory accesses.

Since non-SC behaviors tend to manifest infrequently and unexpectedly, researchers have introduced dynamic analyses that intentionally expose non-SC behaviors allowed under weak memory models [29, 54, 74]. However, these dynamic analyses are limited in the kinds of behaviors they can expose. Figure 5.2 shows an example for which Java’s memory model permits both loads to read the value 1. Existing dynamic analyses cannot expose this behavior because they allow loads to read “stale” values (values stored in the past), but not “future” values (values that will be stored in the future).
Initially $x = 0$

Thread 1:

$x = 7$;

Thread 2:

```
if (x != 0)
    r2 = r1 / x;
```

Figure 5.3: An example program that can generate a divide-by-zero exception under HBMM.

### 5.1.1 More on memory models

Before further explaining the problem, we describe two memory models that are crucial in understanding of the root causes of the problem.

**Happens-before memory model (HBMM).** The following description of HBMM is closely based on prior work [54, 92]. HBMM is an easy-to-understand memory model that provides weak but defined semantics for executions with data races. HBMM limits the values that a load can return according to the *happens-before* relation [77], denoted as $\rightarrow_{hb}$.

A load operation $r$ may return the value written by any store $w$ to the same location, if and only if the following properties hold true:

1. $r \not\rightarrow_{hb} w$ (i.e., $w$ happens-before or is concurrent with $r$).

2. There is no intervening store $w'$ to the same memory location such that 

   \[ w \rightarrow_{hb} w' \rightarrow_{hb} r. \]

HBMM thus still permits various behaviors in which memory accesses appear to execute in an order other than program order, i.e., non-SC behaviors. HBMM allows assertion violations in the two programs in Figures 5.1 and 5.2 (page 106). As another example, HBMM allows a divide-by-zero exception in Figure 5.3.

107
If a load reads from the latest store to a variable, then the behavior is SC. If a load reads from an earlier store, then we say it reads a stale value. If a load reads from a store that has not yet happened, then we say it reads a future value. (Admittedly, concepts such as “latest” and “before” are not well defined in a concurrent execution in which operations are not ordered by happens-before. However, these concepts are well defined in the context of a dynamic analysis that observes conflicting operations to each variable in some global order.) HBMM permits loading both stale and future values. The failing behaviors in Figures 5.1 and 5.3 can be produced by using stale values. However, to produce failing behavior in Figure 5.2, future values are needed.

Furthermore, future values can sometimes produce behaviors different from those produced by stale values. For example, in Figure 5.4, using stale values can cause non-termination, while only future values allow the assertion to fail.

An important caveat of HBMM is that it does not guarantee SC for data-race-free executions—the crucial guarantee mandated by DRF0. HBMM is thus not strictly stronger than DRF0, rendering HBMM unsuitable as a language memory model.\(^\text{14}\) Figure 5.5 shows an example of non-SC behavior allowed by HBMM but not by DRF0. This program is data race free because every SC execution only executes loads. However, under HBMM, each load can speculatively return 1, diverting the control paths to store 1 to x and y, justifying the initial speculative loads.

**Java memory model.** JMM is a strictly stronger memory model than both DRF0 and HBMM [92]. It not only enforces SC for data-race-free executions, but it tries to prohibit

\(^{14}\)Conversely, DRF0 allows arbitrary behavior for racy executions and is thus not strictly stronger than HBMM.
Initially $x = y = 0$

Thread 1:

\[ r = x; \]
\[ y = 1; \]

assert $r = 0$

Thread 2:

\[ \text{while} \ (y == 0) \ { \} \]
\[ x = 1; \]

Figure 5.4: Using stale values, the execution may not terminate. Using future values, the assertion can fail.

Initially $x = y = 0$

Thread 1:

\[ r1 = x; \]
\[ \text{if} \ (r1 == 1) \]
\[ y = 1; \]

assert $r1 = 0$ \&\& $r2 = 0$

Thread 2:

\[ r2 = y; \]
\[ \text{if} \ (r2 == 1) \]
\[ x = 1; \]

Figure 5.5: An example data-race-free program that can fail its assertion under HBMM but not DRF0 [3, 21, 22].

results that could compromise memory and type safety. (JMM introduces a concept called causality to define what behaviors are permitted [92].)

Figure 5.6 shows a canonical example [22, 92, 128] of behavior that JMM prohibits but HBMM (and DRF0) allow. HBMM permits an execution in which each load reads 42. This execution is possible as follows: each load’s value is justified by a store on the other thread, which in turn is justified by the load on the same thread. To see why this behavior might conceivably happen, consider a compiler optimization that modifies each thread’s code to speculatively use a predicted value (e.g., 42) at each load, and then checks the value after the store.
Initially \( x = y = 0 \)

Thread 1:
- \( r_1 = x; \)
- \( y = r_1; \)

Thread 2:
- \( r_2 = y; \)
- \( x = r_2; \)

\text{assert } r_1 \neq 42

Figure 5.6: An example out-of-thin-air result [22, 92, 128].

\textbf{Out-of-thin-air results. } Prior work refers to behaviors such as Figure 5.6 as \textit{out-of-thin-air} (OOTA) results. Prior work has not generally agreed on what constitutes an OOTA result [22,92,128]. In this paper, we reuse the following informal definition of OOTA results: “results that can be justified only via reasoning that is in some sense circular” [22]. Under this definition, Figure 5.5’s non-SC behavior and Figure 5.6’s assertion failure are OOTA results.

Figure 5.7 shows another OOTA example from prior work [128]. JMM only permits executions in which \( r_2 = y \) sees 0. However, HBMM additionally permits executions in which \( r_2 = y \) sees 1. To see why this behavior is possible, suppose that the loads of \( x \) and \( y \) see the value 1. The resulting execution (racy) stores 1 to \( x \) and \( y \), justifying the value seen by the initial loads.

The OOTA behavior in this example actually happens in commercial JVMs [128]. A JVM’s just-in-time, optimizing compiler can eliminate the redundant load of \( y \) at line 5, replacing it with \( r_3 = 1 \). This transformation in turn allows \( x = 1 \) on both control paths, which in turn allows \( r_2 \) to load a value of 1 from \( y \).

We make the following observation: \textit{in order to produce OOTA results, an execution must use future values} (e.g., the OOTA results in Figures 5.5, 5.6, and 5.7 rely on future values). Other unexpected and counterintuitive, yet JMM-compliant behaviors, such as the
Initially \( x = y = 0 \)

Thread 1:
1. \( r_1 = x; \)
2. \( y = r_1; \)

Thread 2:
3. \( r_2 = y; \)
4. \( \text{if} \ (r_2 == 1) \) {
5. \( r_3 = y; \)
6. \( x = r_3; \)
7. } \( \text{else} \ x = 1; \)

assert \( r_2 == 0 \)

Figure 5.7: An example program in which compiler transformations can violate JMM [128].

assertion-violating behavior in Figure 5.2, also require future values. Since real-world JVMs neither conform to the JMM nor prevent OOTA results, it is useful to expose all possible but unexpected results due to future values, whether or not they are OOTA, as long as they conform to both HBMM and DRF0.

### 5.1.2 Exposing Weak Memory Model Behaviors

Despite much effort, data races are widespread. By developing and evaluating dynamic analyses that expose weak memory model behaviors [29, 54, 72, 74, 103, 124], researchers have demonstrated that many real data races lead to harmful behaviors. However, existing dynamic analyses have not exposed the full range of possible behaviors—particularly behaviors due to future values, which are uniquely difficult to expose. *Adversarial memory* (AM) is one such analysis, which we compare against and present in Section 5.3. One contribution of our work is to broaden the exploration of what kinds of harmful behaviors are possible due to data races.
Initially \( x = y = 0 \)

Thread 1:
\[
    r1 = x; \\
    y = 1; \\
\]

Thread 2:
\[
    r2 = y; \\
    \text{if } (r2 == 0) \\
    x = 1; \\
\]

assert \( r1 == 0 \land r2 == 0 \)

Figure 5.8: An assertion failure is not possible under Java’s memory model. Dynamic analysis must be careful not to allow an execution in which the assertion fails.

**Exposing behaviors due to future values.** Since AM and other dynamic analyses simulate behaviors due to stale values only [29, 54, 74], they cannot expose behaviors due to future values (e.g., Figures 5.2 and 5.4).

Exposing effects due to future values (as opposed to stale values) presents a unique challenge: if a load reads a future value (i.e., a value that is expected to be stored in the future), this load operation can “change the future,” so that the anticipated future value is no longer stored in the future, leading to an outcome not permitted by weak memory models for safe languages such as Java. For example, suppose a thread’s store is control-dependent on the value of a load, as in Figure 5.8. Then the assertion-violating outcome is impossible under the Java memory model. Thus, special challenges for using future values are (1) how to predict future values that are likely to be stored in the future and (2) how to validate that speculatively used future values are in fact stored in the future.

This work seeks to expose erroneous behaviors due to future values on data races, such as in Figure 5.2, without exposing behaviors that are not permitted under DRF0 or HBMM, such as the assertion failures in Figures 5.8 and 5.5. Notably, we argue that exposing OOTA behaviors such as in Figures 5.6 and 5.7 is worthwhile since (1) JVMs actually allow some OOTA behaviors, and (2) it is still unclear what behaviors JMM should allow or forbid.
To the best of our knowledge, this project introduces the first dynamic analysis that uses future values and evaluates the effects of using future values in real application executions. Unlike model checking techniques, which explores many executions exhaustively and generally do not scale to large, long-running programs, our technique exposes behaviors due to future values within a single execution.

Figure 5.9 illustrates the behaviors allowed by several memory models, compared with behaviors exposed by prior dynamic analyses and targeted by this paper’s analysis. Neither DRF0 nor HBMM is a subset of the other, as Section 5.1.1 explained. JMM permits behaviors that are a strict subsect of the intersection of DRF0 and HBMM. Typical JVMs do not conform to JMM, and existing analyses for Java programs can only expose a subset of
behaviors allowed by JMM. Our goal is to expose not only behaviors allowed by the JMM, but also behaviors allowed by the intersection of DRF0 and HBMM.

5.2 Preliminaries: Common Notation

This section introduces notation that we use to present both prior work’s adversarial memory (AM) analysis [54] in Section 5.3 and our prescient memory (PM) analysis in Sections 5.4 and 5.5.

We use the following notation to describe a multithreaded execution:

\( t \) : A thread identifier.

\( x \) : A shared-memory variable.

\( v \) : A value loaded from or stored to a variable.

The analyses are mainly concerned with memory access operations, which are each one of the following:

\( rd(t, x) \) : Thread \( t \) loads from variable \( x \).

\( wr(t, x, v) \) : Thread \( t \) stores a value \( v \) to variable \( x \).

Both the AM and PM analyses rely on the classic vector clock algorithm [53, 97] to track logical time and happens-before relations [77]. The algorithm associates a vector clock with each thread and each synchronization object (in Java, every object and volatile field), and updates these vector clocks at synchronization operations (lock acquire and release, monitor wait, thread fork and join, and volatile accesses). The vector clock algorithm uses the following notations:
\( K \): A vector clock, which maps each thread to an integer [97].

\( K \sqsubseteq K' \): This relationship means that for every thread identifier, \( K \)'s integer is less than or equal to \( K' \)'s integer. The vector clock algorithm ensures that if an event at logical time \( K \) happens before an event at time \( K' \), then \( K \sqsubseteq K' \). Otherwise \( K \not\sqsubseteq K' \).

\( C_t \): Represents thread \( t \)'s current vector clock.

### 5.3 Prior Work: Adversarial Memory

Prior work introduces adversarial memory (AM) [54] to expose a subset of behaviors that are allowed under the Java memory model (Section 5.1.1). AM enables loads to see stale values by buffering an execution’s stores and tracking happens-before relations. Like other existing dynamic analyses [29, 74], AM does not allow loads to see future values.

AM associates a write buffer with every shared variable:

\( W_x \): A write buffer for variable \( x \), which has the following form:

\[
W_x = v_1@K_1 \cdot v_2@K_2 \cdot \ldots \cdot v_n@K_n
\]

where \( n > 0 \). Each pair \( v@K \) denotes that a store of value \( v \) to \( x \) was executed at vector clock timestamp \( K \). The write buffer initially contains only \( 0@\bot \) (where \( \bot \) is the vector clock that maps all threads to 0), representing that the variable is initialized to its default value [84].

Algorithms 10 and 11 show the analysis that AM performs at each program store and load, respectively. At each store, AM appends the current \( v@C_t \) to the write buffer. At each load, AM picks a value from the visible set of values from the write buffer. AM picks a value using a heuristic function \( \text{pick()} \) (definition not shown). Our experiments reuse heuristics...
Algorithm 10

STORE [AM]: $\text{wr}(t, x, v)$

\[ W_x \leftarrow W_x \cdot v \cdot C_t \]

Algorithm 11

LOAD [AM]: $\text{rd}(t, x)$

\[
\begin{align*}
\text{let } v & \leftarrow \text{pick}(\text{visible}(W_x)) \\
\text{return } v
\end{align*}
\]

from prior work, which include picking the oldest value, picking the oldest value that is different from the last returned value, and picking a random value from the visible set [54].

Algorithm 12 defines the function $\text{visible}()$ for computing the visible set. The condition in braces specifies what values in the write buffer are legal for the current load to see. Since every value in the write buffer is the result of a concrete, previously executed store, a load cannot see values from future stores. Thus, the rule only needs to ensure that there is no intervening store. A value $v_i$ is legal to return if there is no subsequent store of $v_j$ ($i < j \leq n$) that (1) happens after the store of $v_i$ and (2) happens before the current load. The returned visible set maintains the same order of the values as they appear in $W_x$.

### 5.4 Prescient Memory

A key limitation of AM and related existing work [29, 54, 54, 72, 74, 103, 124] is the inability to “look into the future” and load a future value (Section 5.1). This limitation means that these analyses cannot expose some behaviors that are allowed under weak memory models. To overcome this limitation, we introduce an analysis called prescient memory (PM), which supports using and validating future values. The behaviors exposed and validated by PM are allowed under the happens-before memory model (HBMM; Section 5.1.1). However, PM as presented in this section can expose non-DRF0 behaviors, by producing
Algorithm 12

Helper function

```plaintext
function visible(W_x)
    return \{v_i | 1 \leq i \leq n \land (\n        \n        )\}
```

non-SC results in data-race-free programs such as Figure 5.5 (from Section 5.1). In contrast, Section 5.5 introduces a PM workflow that exposes non-SC behaviors only for programs with data races.

Since every legitimately loaded future value is the result of a future store, PM performs speculative loads to “guess” a future value. At later stores to the same variable, PM tries to validate each speculative load by checking if any concurrent store (i.e., a store that races with the load) actually stores the same value that the load used.

PM uses the same notation as AM, and it calls the visible() function from Algorithm 12. PM maintains the same state as AM (the write buffer W_x) and the following additional state:

\( S_x \): A speculative read history for variable x. \( S_x \) contains tuples of the form \( \langle K, v \rangle \), which denotes that a load of x at time K used a speculative value v. Initially \( S_x \) is \( \emptyset \).

Algorithm 13 shows the analysis that PM performs at a load. The analysis picks a value that is predicted to be a future value, using the predict() function (line 2), which may elect to return the provided latest value instead of a potential future value. (The “...” in predict()'s parameter list represents additional parameter(s) that could be provided to help prediction. Section 5.5 introduces an additional parameter that encodes dynamic program location.) If the predicted value is not in the visible set, then the load operation is speculative, and the analysis records the load in \( S_x \) (line 4).

Algorithm 14 shows the analysis at a program store. In addition to appending the current \( v@C_t \) to \( W_x \) (line 1), the algorithm checks if the current store validates a previously executed
Algorithm 13  
LOAD [PM]: \( rd(t, x) \)

1: let \( v \leftarrow \text{latest}(W_x) \)  \( \triangleright\) Start with current value
2: \( v \leftarrow \text{predict}(v, \ldots) \)
3: if \( v \not\in \text{visible}(W_x) \) then  \( \triangleright\) Is \( v \) a speculative value?
4: \( S_x \leftarrow S_x \cup \{ \langle C_t, v \rangle \} \)
5: return \( v \)

Algorithm 14  
STORE [PM]: \( wr(t, x, v) \)

1: \( W_x \leftarrow W_x \cdot v@C_t \)
2: for all \( \langle K, v' \rangle \in S_x \) do
3: if \( K \not\subseteq C_t \land v' = v \) then  \( \triangleright\) Is speculative load validated?
4: \( S_x \leftarrow S_x - \{ \langle K, v' \rangle \} \)

Speculative load from \( S_x \) (line 3). The algorithm removes all such matching entries from \( S_x \) (line 4).

An execution is valid only if (and when) all speculative loads have been validated. Algorithm 15 checks at program termination if all speculative loads have been validated. If not, the current execution may not conform to HBMM, and any erroneous behavior it exhibits is not worth investigating.

An execution may fail to terminate normally, by throwing an exception or getting stuck in an infinite loop, as a result of a speculative load. This behavior should be considered to be legal under HBMM if and only if PM can validate all speculative loads (i.e., \( \forall x . S_x = \emptyset \)). Otherwise, the behavior is invalid: it is quite possibly not allowed under HBMM.

It is sometimes possible to determine prior to program termination that an unvalidated speculative load can never be validated. Algorithm 16 shows the logic, which can be invoked at any point during program execution. If a speculative load happened before all threads’ current vector clocks (line 2), any future stores will happen after this speculative load—so
the load will never be validated. In a managed language such as Java, it is convenient to implement Algorithm 16 at garbage collection (GC) time, since (full-heap) GC traverses all shared variables $x$, at which point it can process $S_x$.

### 5.5 Making Prescient Memory Practical

PM as described in Section 5.4 presents two main challenges:

1. It is difficult to make PM efficiently produce a valid execution containing future values for large, real programs. In particular, how should PM choose when and which future values to use, what actions can it take to improve the chances that future values will be validated? Prior work that has used and validated future values has not dealt with this challenge; instead it performs model checking of small programs [106] (Chapter 6).

2. PM can expose behaviors that are possible under HBMM but not DRF0. As a result, PM can expose non-SC behaviors even for data-race-free programs, such as the assertion failure in Figure 5.5.

We address the above challenges using a novel approach called the PM workflow (or simply the workflow) that consists of the following components:
Figure 5.10: Overview of the PM workflow. Dashed lines separate distinct program executions.

- profiling of potential future values;

- predicting which loads should use future values, and which values to use; and

- deterministic replay that helps provide consistent behavior between profiling and predicting runs, while also permitting divergence as needed.

Figure 5.10 illustrates the workflow. The workflow limits analysis to memory accesses involved in data races, identified in separate program execution(s) using a dynamic data race detector, as in prior work that uses stale values [29, 54, 74]. By limiting PM only to accesses involved in data races, the workflow avoids producing non-SC results for data-race-free programs.

The rest of this section describes specific challenges and how the workflow’s components address them.

### 5.5.1 Profiling Potential Future Values

In order to assist PM’s prediction of future values, we introduce a separate analysis called *PM-profiler* that produces a set of promising future values for each executed load.

PM-profiler and PM need a mechanism to identify executed load operations. We introduce the following identifier:
A dynamic program location that uniquely identifies a dynamic load. \( l \) encodes both the thread that executed the load and the dynamic instance of the static instruction.

PM-profiler maintains the following data structures:

\( R_x \) : A concrete read history for variable \( x \). Each element in \( R_x \) is a tuple with the form \( \langle l, K, \{v_1, v_2, \ldots\} \rangle \). The set \( \{v_1, v_2, \ldots\} \) is a non-empty set of all visible values at the load operation. Initially \( R_x \) is \( \emptyset \).

\( G_l \) : The “promising” future value set for a load operation identified by \( l \). \( G \) is the interface between PM-profiler and PM: PM-profiler produces \( G \), and PM’s predict() function uses \( G \) as a read-only dictionary.

PM-profiler’s analysis at a program store, shown in Algorithm 17, identifies promising future values. It checks if the current store can provide a future value for any of the previous loads in \( R_x \) (lines 2–4). If the current store is concurrent with a previous load, and the store provides a value that is distinct from any value in the visible set of the load (line 3), then the analysis records the value of the store in the set of promising future values for the load (line 4).

At a program load, PM-profiler computes and stores the visible set for the load, as Algorithm 18 shows. The analysis records the dynamic program location \( l \), the current time \( C_t \), and the visible set in \( R_x \). It always returns the latest value from \( W_x \); PM-profiler does not try to expose any weak memory model behaviors.

### 5.5.2 Predicting Future Values

We now overview the prediction component of the workflow, which is represented with the call to function predict() in Algorithm 13 (from Section 5.4). In order to use \( G \) (the
potential future values produced by PM-profiler), PM passes \( l \) to \( predict() \); that is, PM’s analysis in Algorithm 13 calls \( predict(v, l) \).

There are two different questions for prediction: which loads should use future values, and which future values should they use? In our experiments (Section 5.7), we find that only the first question matters: most loads with future values have only one future value, and using different future values for loads with multiple future values has little impact on the behaviors that PM can expose.

Thus, \( predict() \) is concerned with choosing which loads with a future value (i.e., loads at \( l \) with non-empty \( G_l \)) should use a future value. (When there are multiple values in \( G_l \), \( predict() \) chooses one randomly.) In general, if more loads return future values, the execution is more likely to exhibit new, potentially erroneous behaviors. On the other hand, using more future values means that the execution is less likely to be able to validate every loaded future value.

Our implementation of \( predict() \) supports the following policies:

**All** : Every load with non-empty \( G_l \) uses a future value. Except for microbenchmarks, this policy almost always leads to validation failures. However, it is useful for exposing

\[ Algorithm 17 \]

**STORE [PM-profiler]: wr\((t, x, v)\)**

1: \( W_x \leftarrow W_x \cdot v@C_t \)
2: **for all** \( \langle l, K, \{v_1, v_2, \ldots \} \rangle \in R_x \) **do**
3:  **if** \( K \not\subseteq C_t \wedge v \notin \{v_1, v_2, \ldots \} \) **then**
4:  \( G_l \leftarrow G_l \cup \{v\} \)

\[ Algorithm 18 \]

**LOAD [PM-profiler]: rd\((t, x, l)\)**

1: \( R_x \leftarrow R_x \cup \{\langle l, C_t, \text{visible}(W_x)\rangle\} \)
2: **let** \( v \leftarrow \text{latest}(W_x) \)
3: **return** \( v \)
behaviors that might be possible if the “right” set of loads were selected to use future values.

Selective: The first $k$ executed loads do not use future values, and the following $m$ executed loads use future values. Skipping $k$ executed loads helps when we have identified that some loads either (1) have future values that PM cannot validate successfully or (2) have future values that can be validated and lead to erroneous behavior, but we want to look for other behaviors later in the execution. Using future values for the next $m$ loads only, increases the chances that all future values will be validated.

Per-site: This policy modifies the previous policy so that prediction applies only to one particular static load site (i.e., static program location that performs a load). By running with this policy separately for each site, we can increase the chances of finding a load that produces a future value that can be validated.

Our evaluation tries various combinations of these in order to find future values that can be validated and to expose erroneous behaviors. We also tried introducing randomness into the above policies, but did not uncover any new behaviors as a result.

Behaviors exposed by the PM workflow. The PM workflow exposes not only behaviors allowed by JMM but also additional behaviors permitted by both HBMM and DRF0, i.e., the behaviors labeled “Our goal” in Figure 5.9 (from Section 5.1). Figure 5.7 (from Section 5.1) shows an example of such behavior. While some of these behaviors likely cannot be exposed by any conceivable JVM optimization, these behaviors are still of interest to developers, particularly since the exact set of possible behaviors that JVM optimizations might allow is ill defined.
Initially $x = y = z = 0$

Thread 1:
- $r1 = z$
- if ($r1 == 1$)
  - $y = 1$
- else
  - $z = 1$
- $r1 = z$
- if ($r1 == 1$)
  - $x = 1$

Thread 2:
- $r2 = x$
- if ($r2 == 1$)
  - $y = 1$
- else
  - $z = 1$

Thread 3:
- $r3 = y$
- if ($r3 == 1$)
  - $x = 1$

assert $r3 == 0$

Figure 5.11: An example program for which the PM workflow can cause the assertion to fail.

The PM workflow allows some strange behaviors that are still DRF0—depending on one’s exact definition of DRF0. Consider Figure 5.11, for which the PM workflow can cause the assertion to fail. PM-profiler identifies the racy store $x = 1$ by Thread 1. However, when PM uses that value at Thread 2’s load, the data race on $z$ would not manifest and Thread 1’s store to $x$ does not execute. Instead, Thread 3’s store to $x$ validates the load.

5.5.3 Fuzzy Deterministic Replay

Multithreaded executions are inherently nondeterministic due to timing-sensitive thread interleavings. This nondeterminism presents two challenges for PM-profiler and PM, which operate on separate executions. First, nondeterminism makes it difficult to match loads across program executions: the mapping $G$ produced by PM-profiler is unlikely to be useful to PM if program execution diverges. Second, nondeterminism makes it less likely that a potential future value from a prior execution will actually be validated by a future store.
Our workflow thus extends *multithreaded record & replay* [24, 80, 137] in order to eliminate nondeterminism between the PM-profiler and PM executions. Deterministic replay helps guide the PM execution to match the PM-profiler execution’s thread interleavings. However, after PM uses a future value, execution may *diverge* from the recorded execution. Nonetheless, we have found that deterministic replay is still useful at this point in order to potentially guide the execution to store (and thus validate) the future value.

In some cases, divergence could cause the deterministic replay mechanism to be unable to continue. For example, suppose thread T2 is waiting for thread T1 to reach a specific execution point in order to ensure deterministic replay. If T1 loads a future value and diverges from the recorded execution, T1 may never reach the point that T2 is waiting for, in which case replay is “stuck.” Instead of failing the execution, PM detects when replay is stuck and proceeds *without being guided by replay*. We refer to this best-effort replay approach as *fuzzy replay*. Fuzzy replay is useful not only for validating future values at upcoming stores, but also for *using* additional future values at upcoming loads after the execution has already diverged.

### 5.6 Implementation

We have implemented AM, PM, and the PM workflow in Jikes RVM 3.1.3 [8, 9], a high-performance Java virtual machine (JVM) that performs competitively with commercial JVMs [16]. Our implementation of the PM workflow builds on existing, publicly available implementations of dynamic data race detection (the FastTrack algorithm [53] implemented in Jikes RVM [16]) and multithreaded record & replay [24]. Our implementation of AM is influenced by a publicly available implementation of AM in Jikes RVM [125].
AM, PM, and PM-profiler modify Jikes RVM’s dynamic compilers to add instrumentation at every memory access identified by the data race detector. The analyses bound the size of each variable’s write buffer \((W_x)\) and read history \((R_x\) for PM-profiler; \(S_x\) for PM) in order to avoid running out of memory. The implementations represent dynamic program location \(l\) as a tuple of the thread, the static site (method and bytecode index), and a per-thread, per-site counter.

We extend the existing record & replay implementation to support our workflow. We extend the record and replay analyses to record and replay synchronization operations (which normally would be ignored [24]), so PM can perform the vector clock algorithm. To support fuzzy replay, we extend replay so that it stops trying to replay if it gets stuck or encounters a replay error. In order to support multiple replay attempts from one recorded execution, we extend the “fork-and-restart” mechanism that the record & replay implementation uses [24].

A limitation of our current implementation is that it can use future values with primitives types but not reference types (i.e., references to objects). This limitation exists because object addresses are not in general the same across record and replay, since the record & replay implementation provides application-level determinism [24]. That said, it should be possible to use reference types by extending our implementation. Objects can be identified uniquely across runs by tagging each object with its thread and a per-thread counter incremented at each allocation (the record & replay implementation already identifies objects this way in order to provide deterministic hash codes [24]), although this approach would seem to require the ability to look up any object based on its tag during replay. Another challenge is that at a load to a reference-type future value, the object may not have been allocated yet. The implementation could address this by eagerly allocating (but not initializing) the object at the load.
5.7 Evaluation

This section evaluates the PM workflow’s ability to expose erroneous program behaviors using future values. In this section, “PM” refers to the PM analysis executing as part of the PM workflow (Section 5.5), not the general form of PM from Section 5.4.

5.7.1 Methodology

Our experiments execute benchmarked versions of real applications: the DaCapo benchmarks, versions 2006-10-MR2 and 9.12-bach (2009) [17] (limited to multithreaded programs that Jikes RVM can run), and fixed-workload versions of SPECjbb2000 and 2005.15

We build a high-performance configuration of Jikes RVM. The experiments run on a system with an Intel Core i5-2500 4-core processor running Linux 2.6.32. (We also tried running experiments on a 32-core machine, but that did not expose any new behaviors.)

For a fair comparison between AM and PM, the experiments only consider fields of non-reference types, since the PM implementation does not currently support reference types (Section 5.6). We execute AM and PM repeatedly for each program, trying out different AM heuristics [54] and PM prediction policies (Section 5.5.2) to see what kinds of erroneous behaviors can be exposed, such as corrupted output, exceptions, and non-termination.

5.7.2 Stale and Future Values Found in Real Applications

Table 5.1 shows the number of distinct fields that can load stale values or potential future values. The result is a union of distinct fields discovered in 5 trial runs of AM and PM-profiler, respectively. We note that the reported set of fields are almost identical from run to run.

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of distinct fields that can load:</th>
<th>future values</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsqldb8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>lusearch6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xalan6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>avrora9</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>jython9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>luindex9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lusearch9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pmd9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sunflow9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>xalan9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pjbb2000</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>pjbb2005</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.1: Statistics of the number of distinct fields (non-reference type) that can load stale or future values for each program.

For most fields that can return a stale value in AM, PM-profiler can detect a future value for some loads to these fields. We did not find any field that can load future values but not stale values. Nonetheless, returning future values sometimes uncovers different, and arguably more destructive erroneous behaviors than returning stale values does, as described in the next subsection.

We have tried different policies and tuned parameters for predict() (Section 5.5.2), in order for PM to produce valid executions that contain future values and may bring about some new behaviors. For each of the potential fields reported by PM-profiler, PM can always produce a valid execution that contains at least one load of a future value. We found that the effects of the policies and their parameters varies from program to program. Some programs may need an aggressive setting that uses many future values to expose a bug (e.g., avrora9), while some programs may need a conservative setting to use few future values to pass validation (e.g., lusearch9, sunflow9).
Program | Field(s) involved | AM Worst erroneous behavior (Observable?) | PM workflow (Observable?)
--- | --- | --- | ---
Figure 5.1 | x, y | Assertion failure (Yes) | None (N/A)
Figure 5.2 | x, y | None (N/A) | Assertion failure (Yes)
Figure 5.3 | x | None (N/A) | None (N/A)
Figure 5.4 | x, y | Divide-by-zero (Yes) | Divide-by-zero (Yes)
Figure 5.5 | None* | None (N/A) | None (N/A)
Figure 5.6 | x, y | None (N/A) | None (N/A)
Figure 5.7 | x, y | None (N/A) | None (N/A)
Figure 5.8 | x, y | None (N/A) | None (N/A)
Figure 5.9 | x, z | None (N/A) | None (N/A)
hsqldb6 | MemoryWatcherThread.keep_running | Non-termination (Yes) | Data corruption (Yes)
hsqldb6 | JavaSystem.memoryRecords | None (N/A) | Performance bug (No)
avrora9 | Transmission.lastBit | Data corruption (Yes) | Data corruption (Yes)
lusearch9 | ThreadLocal.nextHashBase | Performance bug (No) | None (N/A)
sunflow9 | Geometry.builtAccel | Null ptr exception (Yes) | Null ptr exception (Yes)
pjbb2000 | Company.mode | Non-termination (Yes) | Data corruption (Yes)
pjbb2000 | Company.elapsed_time | Data corruption (Yes) | Data corruption (Yes)
pjbb2005 | DomNode.eventDataLock | Data corruption (No) | Data corruption (No)
pjbb2005 | DomEvent.stop | Data corruption (No) | None (N/A)

Table 5.2: Summary of erroneous program behaviors discovered by returning stale or future values. *The program in Figure 5.5 is data race free, so AM and the PM workflow do not instrument any memory accesses.

hsqldb6, avrora9 and pjbb2005 have 5 additional fields that can return stale values but not future values. However, we do not find any erroneous behaviors caused by stale values for 4 of these fields. One field, which is in pjbb2005, can have erroneous behavior due to stale values, as described below.

5.7.3 Exposing Erroneous Behavior

Table 5.2 summarizes erroneous behaviors discovered by our implementations of AM and PM. For completeness, our evaluation includes results for the example programs in Figures 5.1–5.7 and Figure 5.11. For the 12 real programs we evaluated, PM exposes 7 erroneous behaviors. Of these 7 bugs, AM can expose the same behaviors for 4 of them, AM exposes different behaviors for 2, and AM cannot expose the bug for 1. Additionally, AM can expose erroneous behavior for 2 bugs for which PM cannot expose erroneous behavior.
In some cases, the same error manifests differently in AM and PM, e.g., non-termination versus data corruption.

Interestingly, PM exposes erroneous behavior for most of the same bugs for which AM exposes erroneous behavior, even though PM does not use stale values. Our evaluation intentionally compares analyses with non-overlapping functionality: AM uses only stale values, while PM uses only future values. A more powerful analysis would ideally combine AM and PM in order to load both stale and future values.

In the evaluated real programs, PM does not expose any out-of-thin-air (OOTA) results. Nonetheless, researchers and practitioners could use the PM workflow to identify OOTA behaviors, including controversial and/or JMM-violating behaviors. Any real-world evidence of such behaviors would inform future revisions to language specifications.

Our experiments detect a few stale and future values beyond those reported in Table 5.2. For 5 fields (2 in hsqldb6, 2 in avrora9, and 1 in sunflow9), AM detects stale values but cannot expose erroneous behavior. For the field in sunflow9, PM-profiler detects future values, which PM can use and validate, but it cannot expose erroneous behavior. Other than these cases and the cases in Table 5.2, there are no fields for which AM detects stale values or PM-profiler detects future values (including future values that PM cannot validate).

**Microbenchmarks.** The table shows that PM and AM behave as expected for the microbenchmarks corresponding to Figures 5.1–5.7 and Figure 5.11. Although the general form of PM presented in Section 5.4 can expose erroneous behavior for Figures 5.5 and 5.6, the PM workflow cannot.

**hsqldb6.** This database management system has a thread that continuously monitors the application’s memory usage and uses a boolean flag MemoryWatcherThread.keep_running.
as a termination condition. Threads access this variable racily. Returning a stale value can prevent the thread (and consequently the whole program) from terminating. Returning a future value can cause the thread to terminate early and corrupt memory usage statistics. By default, the benchmarked version of the program does not output the statistics, but we have modified it to do so, making the data corruption visible.

JavaSystem.memoryRecords is a counter that the program increments at certain memory operations. The program periodically checks the counter to decide if it should trigger garbage collection (GC):

```java
if (memoryRecords > n) {  // n is a run–time constant
    memoryRecords = 0;
    System.gc();
}
```

Returning a stale value can trigger GC less frequently, but triggering GC is unnecessary since the JVM does it automatically. (Furthermore, JVMs are permitted to ignore System.gc() calls [84].) Returning a future value can trigger GC more frequently; PM is able to successfully use and validate future values. In theory, repeatedly using a large future value could cause a performance bug by triggering GC frequently. However, we have been unable to produce PM executions that use future values that can be validated but are also large enough to cause noticeable slowdowns.

avrora9. This program is a simulator for an embedded microcontroller. Transmission.lastBit is a long field that indicates the end-byte position of a simulated radio transmission. The program uses this field to compute a list of intersecting transmissions and other simulation metrics:
List getIntersection (long bit) {
    List it = null;
    synchronized (medium) {
        Iterator i = medium.transmissions.iterator();
        while (i.hasNext()) {
            Transmission t = (Transmission) i.next();
            if (it == null) it = new LinkedList();
            if (bit >= t.firstBit && bit < t.lastBit) {
                it.add(t);
            }
        }
        return it;
    }
}

Loading from this field (line 8) can return stale values and future values that can be validated. In both cases, the values lead to an incorrect list and corrupt the resulting metrics. However, this corruption is infrequent in our experiments. In 500 trials each for AM and PM, we found that AM and PM corrupted output in 23 trials (4.6%) and 10 trials (2.0%), respectively. It is easier to expose this bug using AM: all AM executions are always legal under HBMM (and JMM, in fact). For PM, more than half of the 500 executions failed to validate every future value, making them invalid even though they corrupted output in some cases.

lusearch9. This program uses the lucene indexing and search library to perform text search. The program uses a field ThreadLocal.nextHashBase, which is part of GNU Classpath, the Java library implementation used by Jikes RVM. The library uses the counter to initialize a final field, hashCode, for each new ThreadLocal instance. The method that increments nextHashBase is synchronized but (erroneously) not static. We note that this bug should be attributed to GNU Classpath, not the lusearch9 benchmark or lucene library.
Two object instances can share the same hash code value, as long as the `hashCode` field remains constant after initialization, which could lead to a performance bug by increasing the chances of hashing collisions. However, `ThreadLocal` is used infrequently in this program, so a performance bug is not observable. Future values could in theory lead to a performance bug by using the same future value for many loads, but our experiments cannot successfully validate executions in which many loads use future values.

**sunflow9.** The program uses a double-checked locking pattern to lazily initialize the shared reference `Geometry.accel` (code simplified from the original):

```java
if (builtAccel == 0) {
    synchronized(this) {
        if (builtAccel == 0) {
            accel = new ...;
            builtAccel = 1;
        }
    }
    accel.intersect (...);
}
```

The accesses to `accel` and int field `builtAccel` are racy because the program fails to declare `builtAccel` as `volatile`. Returning a future value of `1` for `builtAccel` at line 1, can trigger a null pointer exception (NPE) at line 9. (AM is also able to expose this bug if it instruments accesses to the reference-type field `accel`, which can return a stale value of `null` at line 9.)

The following figure shows an interleaving that PM can produce by using future values. The arrow connects the racy accesses that load and later store the future value `1`. 
if (builtAccel == 0) {
    ...
    // Not executed
}  
accel.intersect(...); // NPE

if (builtAccel == 0) {
    synchronized(this) {
        if (builtAccel == 0) {
            accel = new ...;
            builtAccel = 1;
        }
    }
    accel.intersect ( ... );

Accesses to another int field Geometry.builtTess use a similar racy double-checked locking pattern. However, returning a stale value or a future value on this field does not lead to any erroneous behavior that we could detect.

pjbb2000. This program is an artificial benchmark that simulates the backend of a business server. It uses a field Company.mode to maintain the state of a Company object. The program uses the following unsynchronized load of mode to decide whether to update statistics data:

if (company.mode == Company.RECORDING)
    myTimerData.updateTimerData(txntype, txntime);

PM returns a future value of Company.RECORDING at this load, leading the program to take the true branch, which should not be taken until later in the execution, corrupting reported statistics.

Returning a stale value cannot trigger this data corruption, because the value Company.RECORDING does not exist in the set of stale values. Nonetheless, using a stale value for a different program load of mode can lead to non-termination. For this other load, PM-profiler detects no future value.
For another field `Company.elapsed_time`, both stale and future values lead the program to report an incorrect timing value, corrupting the output statistics. (Table 5.2 thus reports the erroneous behavior as “observable.” However, the behavior may be hard to observe in practice because the program is an artificial benchmark targeting performance testing and does not have a clear specification for correct output, which is nondeterministic from run to run.)

pjbb2005. This artificial benchmark invokes the following code in XML processing libraries that are part of the GNU Classpath implementation. `DOMNode.eventDataLock` is a `static boolean` field that helps to enforce mutual exclusion. The program minimizes allocations of `DomMutationEvent` objects using code shown in Figure 5.12.

Consider the following scenario. One thread initializes a `DomMutationEvent` object (line 17) and dispatches it to a target node object (line 18). Instead of allocating a new `DomMutationEvent` object every time, the code tries to reuse the shared “scratch” object referenced by `m` (lines 7–13). The `eventDataLock` field indicates if `m` is currently used by a thread. In a sequentially consistent (SC) execution, lines 17–18 execute atomically when threads reuse the shared object `m`.

However, “releasing” `eventDataLock` on line 21 is racy. This permits the load of the field on line 8 to return `false` even if the SC value would be `true`. As a result, two threads can simultaneously use the shared object on lines 17–18, violating mutual exclusion. Using either a stale value or a future value can trigger this behavior. The following illustration shows an interleaving with a future value, with the arrow connecting the load and store that use and store the future value of `false`, respectively.
static boolean eventDataLock = false;
static Object lock = new Object();
static DomMutationEvent m = new DomMutationEvent();
void insertionEvent(DomNode target) {
    boolean doFree = false;
    DomMutationEvent e = null;
    synchronized(lock) {
        if (!eventDataLock) {
            eventDataLock = true;
            doFree = true;
            e = m;
        }
    }
    if (e == null) {
        e = new DomMutationEvent();
    }
    e.initialize (...);
    target.dispatchEvent(e);
    if (doFree) {
        e.clear ();
        eventDataLock = false;
    }
}
The `dispatchEvent()` method (called from line 18 in Figure 5.12) accesses another field, `DomEvent.stop`. As a result of the racy “release” of `eventDataLock`, loads to `DomEvent.stop` can return a stale value, prematurely ending a traversal of a `DomNode` array and possibly corrupting data. PM-profiler does not detect any future values for this field.

For both fields, we have been unable to detect any visible effect from the output of `pjbb2005`, even though using a stale or future value can violate mutual exclusion and corrupt memory states. We suspect that since this benchmark is designed for performance testing alone, it lacks sensitivity to this data corruption. In any case, these bugs (like the `lusearch9` bug) should be attributed to GNU Classpath, not to `pjbb2005`. 

137
Figure 5.13: Run-time overhead of AM, PM-profiler, and PM. Each bar is the median of 10 trials. The intervals are 95% confidence intervals centered at the mean. Overheads exceeding 700% are labeled using two significant figures.

5.7.4 Run-Time Performance

This section measures the run-time overhead added by PM, compared with AM. We run configurations of AM and the PM workflow that do not use any stale or future values; PM-profiler still records potential future values, and PM simulates the cost of using future values by recording them in the read history $S_x$.

Figure 5.13 shows the run-time overhead that each analysis adds over execution on the unmodified JVM. The average overhead of AM is 76%. PM-profiler incurs almost 600% overhead on average, while PM incurs 390%. We find that less than one-third of PM-profiler and PM’s overhead comes from the record and replay analyses, respectively.
PM-profiler and PM perform significantly more work than AM and thus add substantially more overhead. PM-profiler adds more overhead than PM since only PM-profiler tracks the concrete read history $R_x$ for each variable $x$. AM and PM add overhead proportional to the frequency of instrumented (racy) accesses, so measured overhead varies significantly across the evaluated programs. We have not endeavored to optimize the implementations, which for convenience use inefficient patterns (e.g., heavy use of containers with boxed primitives).

5.8 Contributions and Impact

Prescient memory (PM) enables programs to return future values at load operations via a speculation-and-validation approach. We introduce a novel, practical workflow that incorporates profiling, prediction, and fuzzy replay to help PM use future values successfully in large, real applications. Our evaluation demonstrates that this approach effectively exposes previously unknown erroneous behaviors due to future values.

**Contributions.** This work makes the following intellectual and empirical contributions:

- PM is the first dynamic analysis that exposes weak memory model behaviors due to future values in large, real applications. In order to enable PM to expose these behaviors without exhaustive exploration, we introduce a novel approach called the PM workflow that incorporates three components: profiling, prediction, and fuzzy replay. Our evaluation shows that this approach is in fact useful for using future values successfully.

- Our evaluation shows that legal uses of future values exist in real applications and that they can lead to harmful behaviors. Future values alone (i.e., without using any stale values) can often trigger the same bugs that stale values trigger—but the future-value
behaviors are sometimes different and more destructive. These results motivate our approach’s utility for exposing previously unknown program behaviors.

• An existing line of research shows that seemingly “benign” data races are in fact harmful [29, 54, 72, 74, 103, 106, 124]. By exposing real, destructive behaviors due to future values, our work advances the state of the art in this area.

• Existing language memory models still have difficulty defining what program behaviors should be allowed for an execution with data races [3, 22, 128]. Our approach provides an opportunity to explore this gray area in large, real programs; real-world evidence of controversial examples would inform and influence future language specification revisions.

Despite years of study and research on data races, software developers and analyses still have limited understanding of possible behaviors for racy executions, and language memory models are unable to define semantics of data races satisfactorily. PM overcomes a key limitation of existing dynamic analyses that are unable to use future values, advancing the state of the art in practically exposing behaviors possible under weak memory models. By exposing unknown behaviors of real racy executions, PM helps software developers and researchers understand the full effect of data races. As a result, software developers will be more convinced to write data-race-free programs and fix existing races, and researchers will have more evidence to design future memory models and language specifications. These efforts will improve the reliability and dependability of software and systems.
Chapter 6: Related Work

Chapter 2 covers prior work in data race and memory models. Chapter 3 to 5 includes existing techniques in dependence tracking, record & replay, and exposing erroneous behaviors. This section describes other prior work related to proposed approaches in this thesis.

**Strong memory models.** Researchers have argued that languages and hardware must provide stronger end-to-end memory models to avoid impossibly complex semantics for executions containing data races [3, 20, 35, 90]. Strong memory models eliminate erroneous behaviors allowed by weak memory models, and they constrain behaviors in ways that programmers already (mistakenly!) expect under DRF0-based memory models [109, 125].

Much research has targeted SCMM [7, 59, 78, 82, 83, 90, 94, 95, 115, 129, 131, 132, 134]. However, it seems to be inherently expensive to enforce end-to-end SC, which constrains compiler and hardware reordering. And yet SC is not a particularly strong model: it permits many possible interleavings that programmers generally do not expect.

A promising alternative to SC are memory models based on ensuring *serializability* (atomicity) of dynamically executed regions of code, called *region serializability* (RS) [16, 90, 94, 109, 125–127, 131]. Notably, serializability of SFRs is a strong memory model that enforces atomicity of SFRs for all executions. However, existing approaches for providing...
serializability of SFRs are impractical: they either rely on complex custom hardware [90] or slow execution by two or more times [16, 109].

We note that a widespread adoption of a strong memory model will significantly affect the design and application of runtime support. Runtime support may not need to track cross-thread dependences in order to provide its desired functionality, thus ameliorating the performance issue. For example, for executions that always conform to serializability of SFRs, a record & replay system could just record the commit order of SFRs. Some runtime support such as data race detectors, transactional memory, and analyses that expose errors may arguably have less significance, since racy executions would have well defined semantics. While a strong end-to-end memory model seems the ultimate solution to today’s issues on data races and runtime support, it is still too early to tell which path the industry will eventually take. It is unrealistic to rely on a strong memory model to address current challenges in runtime support.

**Program locks.** Program locks face similar tradeoffs as pessimistic versus optimistic tracking. Notably, *biased locking* avoids atomic operations for repeated lock acquisitions by the same thread, requiring coordination when another thread acquires the lock [30, 75, 119]. A biased lock typically falls back to an unbiased lock after triggering coordination once.

**Adaptive mechanisms.** Prior work has used adaptive techniques to combine different kinds of synchronization. Usui et al. use online profiling and a cost–benefit model to adaptively choose between lock-based mutual exclusion and software transactional memory (STM) for enforcing atomicity of critical sections [136]. Abadi et al. present an STM that adaptively changes how it detects conflicts for non-transactional accesses, depending on whether transactions access the same objects as non-transactional code [2]. Dice et al. build
a runtime library that supports adaptive lock elision using hardware transactional memory (HTM) and optimistic software execution [44]. Ziv et al. formalize a theory for correctly composing different concurrency control protocols in programs [151].

**Tracking dependences using commodity hardware.** Intel’s recently introduced Haswell architecture provides *restricted transactional memory* (RTM): best-effort TM support with an upper bound on shared-memory accesses in a transaction [145]. Recent work finds that an RTM transaction must be expanded to replace at least 3–4 atomic operations, in order to amortize the overhead of a transaction [96, 117, 145]. Other work modifies a dynamic data race detector to replace *pessimistic tracking* with RTM-based tracking, which requires combining several critical sections in order to overcome the costs of RTM [96]. Prior results suggest that optimistic tracking is likely to outperform RTM for non-conflicting accesses by avoiding atomic operations altogether, while a hybridization of pessimistic and optimistic tracking is likely to perform best for a mix of high- and low-conflict accesses.

**Deterministic runtime and languages.** Researchers have proposed approaches to execute multithreaded programs deterministically, as an alternative to record & replay [11, 13, 18, 41, 43, 86, 108, 116]. These approaches either enforces determinism through custom runtime [13, 41, 43, 86, 108], or introduces new programming languages [11, 18, 116]. Runtimes that provide determinism suffer from limitations similar to those in record & replay approaches. They either rely on custom hardware [43], ignore data races [108], or add high overhead [13, 41]. Dthreads avoid data races and provide determinism by mapping threads to processes in order to isolate memory space, and it merges memory state at synchronizations [86]. However, this approach will not scale well to programs with ad-hoc...
or fine-grained synchronizations. Approaches that introduce new languages for determinism require rewriting programs, which is impractical in production [11, 18, 116].

**Model checking.** Prior work also employs model checking and verification techniques to explore concurrent program behaviors, including effects due to data races under weak memory models [27, 28, 62, 69, 99, 106]. These techniques typically offer better theoretical guarantees and greater coverage than dynamic analyses. However, model checking typically suffers from state-space explosion for realistically sized programs. Thus, these tools commonly target small portions of code such as concurrent data structures and core algorithms, instead of entire large applications.

*CDSChecker* is a model checker that exhaustively explores program behaviors allowed by the C/C++ memory model [106]. It only considers C/C++ atomic variable accesses, since racy accesses on ordinary C/C++ variables have undefined semantics. CDSChecker supports returning both stale and future values for atomic variable loads, which are permitted by the C/C++ memory model [21] with constraints similar to those in the happens-before memory model (HBMM; Section 5.1.1). It would be infeasible to extend CDSChecker’s exhaustive approach to large, real programs, particularly for a safe language such as Java, in which every potentially racy access can have weak memory model behavior. CDSChecker’s evaluation does not report any new behaviors from future values (i.e., no behaviors not already possible by using stale values).
Chapter 7: Conclusion

This section concludes the thesis with a summary of work presented and a discussion for their impact.

7.1 Summary

Despite decades of research, data race remains one of the most problematic concurrency bug. Data races are widespread in modern parallel software, but unfortunately there is no known satisfactory solution that fixes, eliminates, or tolerates all data races. Dynamic analyses for concurrency correctness must be efficient and capable of dealing with executions with data races, in order to be practical for production use.

We present three dynamic analyses that address performance and coverage limitations in prior approaches for dealing with racy executions:

• **Chapter 3** presents hybrid tracking, an efficient framework for tracking cross-thread dependences that overcomes limitations of pessimistic and optimistic tracking. Hybrid tracking is suitable for building runtime support that targets diverse workload.

• **Chapter 4** presents RegPlay, an efficient, software-only multithreaded record & replay algorithms that guarantees replay determinism for all executions. RegPlay applies novel insights to avoid recording unnecessary dependences, in order to achieve low time and space overhead for record.
• **Chapter 5** presents prescient memory, a novel dynamic analysis that exposes a behaviors of data races due to loading future values. Prescient memory is able to expose unexpected erroneous behaviors in large, real applications, overcoming a key coverage limitation of existing dynamic analyses for exposing program bugs.

7.2 Impact and Meaning

Parallel software and systems are ubiquitous in the modern world. They are constantly running everywhere, ranging from broad areas such as scientific discovery, finance, health, education, and communication, to small devices such as smartphones, watches, and household appliances. Software bugs and failures can cause detrimental effects to the society, sometimes even threaten human life.

However, there still lacks effective tools that help programmers to develop and debug parallel software. Many existing approaches targeting concurrency correctness properties are impractical. This dissertation proposes software-only techniques that are efficient and practical. In particular, all proposed techniques effectively support executions with data races, addressing challenges in terms of performance and coverage.

The proposed techniques advance the state of art in their respective areas. Hybrid tracking is the first approach that combines pessimistic and optimistic tracking effectively and efficiently in the context of design and implementation of runtime support. RegPlay is the first multithreaded record & replay algorithm that preserves replay determinism and supports both offline and online replay by only logging write–write and write–read dependences. Hybrid tracking and RegPlay demonstrate new directions to improve the performance of dependence tracking in generalized and analysis-specific ways, respectively. Prescient memory is the first dynamic analysis that can expose behaviors due to future values.
caused by data races in large, real programs, extending the scope of program behaviors that
software-only approaches can explore.

Consequently, this dissertation will promote the adoption of analyses and runtime for
parallel software. In the long term, these tools will improve reliability, scalability of software,
and productivity of software development, which will have far-reaching impact that benefits
every area that relies on parallel computing systems.
Bibliography


156


