# Software and Hardware Support for Data-Race Exceptions

Benjamin P. Wood

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

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Dan Grossman, Chair

Luis Ceze, Chair

Zachary Tatlock

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

## Abstract

Software and Hardware Support for Data-Race Exceptions

Benjamin P. Wood

Co-Chairs of the Supervisory Committee:
Associate Professor Dan Grossman
Computer Science & Engineering

Associate Professor Luis Ceze
Computer Science & Engineering

Some researchers have proposed data-race exceptions to mitigate the ill effects of data races in shared-memory multithreaded programs. Data-race exceptions make every data race an explicit fail-stop error at run-time. Implementing data-race exceptions naturally requires accurate dynamic data-race detection with low performance overhead, yet existing data-race detectors compromise either accuracy or performance. Hardware data-race detectors solutions are fast, but inaccurate. Accurate software data-race detectors slow execution by several times. This dissertation presents three new systems to bring accurate and fast language-level data-race exceptions closer to feasibility.

*Race Detection in Software and Hardware (RADISH)* accelerates an accurate software data-race detection algorithm by mapping common cases to highly-optimized hardware support. By falling back to software support in rare cases, RADISH maintains full accuracy for low-level programs while achieving good performance. We show RADISH's accuracy via its equivalence to a canonical accurate software algorithm for data-race detection.

*Low-level Abstractable Race Detection (LARD)* virtualizes accurate low-level data-race detectors, such as RADISH, to support accurate data-race detection for high-level languages. Experimental evaluation shows that existing low-level data-race detectors are inaccurate on high-level programs in practice, while our LARD implementation is accurate and preserves the performance of the low-level data-race detector.

*Fast Instrumentation Bias (FIB)* is a cooperative synchronization protocol designed to reduce the overheads of pure-software accurate dynamic data-race detection. *Analysis barriers*—the code inserted before each memory access in the program to check and update analysis metadata—may execute concurrently. If barriers are not atomic, they may fail to detect true data races. Existing implementations either allow non-atomic barriers, sacrificing guaranteed accuracy, or employ pessimistic synchronization to ensure barrier atomicity and analysis accuracy. FIB exploits analysis invariants to guarantee barrier atomicity with no synchronization in the common case, at the cost of expensive synchronization in rare cases. Experimental evaluation shows that FIB is faster than a highly optimized conventional implementation of barrier atomicity on several benchmarks and slower on others, varying with the rate of updates to shared data. Conservative dynamic thread-escape analysis can lower overheads of both implementations while maintaining accuracy.

# Table of Contents

# List of Figures

# List of Tables

## Acknowledgments[1]

The work described in this dissertation was undertaken with several collaborators. Joe Devietti led the Radish project, where he and I enjoyed a 2-to-1 adviser-to-student ratio working with Karin Strauss, our advisers Luis Ceze and Dan Grossman, and Shaz Qadeer. The breadth of experience and insight represented in this group was exciting, as was publishing a paper at a computer architecture conference that referred to a companion technical report with *proofs*, sketchy as they may have been. On the Lard project, Dan and Luis offered great support, repeatedly championing an idea I worried was too obvious. On the Fib project, Dan and I have benefited from the insights and experience of collaborators Man Cao and Mike Bond, who designed and engineered Octet, the most related prior work.

More broadly, in six years at UW CSE, I have interacted with a host of insightful individuals who made me the computer scientist I am today.

My advisers Dan Grossman and Luis Ceze brought complementary perspectives to their co-advising that helped build the signature Sampa culture of cross-stack thinking while supporting the goals of the individual students they advised. They proved patient in their attempts to focus a student perpetually distracted by unrelated pursuits, and reacted with easygoing laughter (and occasional delight) when said student poked fun at them and the grad school experience. Dan, with his knack for the right mix of rigor, clarity, and a little humor, influenced the way I think and communicate (even if I still wave my hands a bit too often), whether for research or teaching. Luis, with his seemingly unbounded and unbridled enthusiasm, threw more inspired and zany ideas at me than I could dodge, eventually turning

[1]The section's title is an acronym describing the peculiar genre known as dissertation acknowledgments: ***A****wkwardly* ***c****ooing* ***k****ey* ***n****ames* ***o****r* ***w****(e)arily* ***l****isting* ***e****ach* ***d****ue* ***g****ratitude* ***m****akes* ***e****ffusive* ***n****otes* ***t****urn* ***s****appy.*

me from ignoring hardware to dabbling occasionally in computer architecture. I must also thank Dan for an ancillary benefit of his advising: His foresight in taking a couple "long walks" in his college days ensured that he and Luis were not only on board when I decided to disappear to walk for a few months, but piloted a version of the LARD paper through author response and resubmission so I could keep walking.

I also benefited from mentorship by other faculty during my time at UW. Gaetano Borriello offered mentorship and support in my steps from CSE 351 TA to instructor to faculty job applicant and beyond. At my talks at UW, Mark Oskin always asked big questions that I never quite understood, but that have later deepened my own understanding of my research. Zach Tatlock offered enthusiastic support and fresh insights on presentation, especially in hasty preparation for job talks. I somehow conned him into serving on my committee during his first year as a professor. Scott Hauck served as graduate school representative at my general exam despite the peculiar proposed completion schedule and Nick Boechler stepped in for my final exam on short notice to help solve a difficult set of schedule constraints. Hal Perkins shared candid discussions on teaching careers.

Sebastian Burckhardt, along with Daan Leijen and Manuel Fähndrich, exposed me to new perspectives, insights, and values in research, during a brief stint at Microsoft Research.

Steve Freund played a major role in my entry into computer science research and grad school. My first research project was on dynamic data-race detection with him seven years ago. (Apparently my fascination by this topic remains.) Steve's relaxed mentorship on points from academic life balance to research detail has been invaluable ever since.

Lindsay Michimoto was a source of wisdom, administrative magic, and encouragement throughout grad school. I remained blissfully ignorant of grad school finance due to Lindsay, Mel Kadenko, Julie Svendsen, Lisa Merlin, Joel Cohn, Jan Harrison, and others, while enjoying the support of a departmental Anne Dinning and Michael Wolfe Fellowship and an ARCS Foundation Fellowship.

The students of SAMPA made collaboration a blast. Joe Devietti and I had a knack for

writing papers that took program committees at least a few tries to appreciate.[2] Maybe this is because we consistently omitted our best ideas, like representing 9 states in 3 bits via an enchanting theory of half-bits, or deploying the title, *Dr. Dish Towel: Data-Race Detection In Software and Hardware That's always On With Excellent Latency*. Brandon Lucia's creative enthusiasm and shared enjoyment of terrible puns kept me on my toes. Without Adrian Sampson's level-headed and insightful assistance, my first conference paper would not have happened so smoothly. I later learned a lot from Adrian's ability to deliver compelling and clear presentations. Jacob Nelson's masterful management of our group infrastructure saved me countless headaches. Discussions with Jacob and Nick Hunt revealed reassuring shared perspectives on some peculiarities of the value systems of academia. Hadi Esmaeilzadeh demonstrated that a brilliant researcher could be downright friendly. Finally, post-Sampa-meeting afternoon diversions with Brandon, Adrian, Joe, Jacob, ~~Tom Bergan~~, Emily Fortuna, Brandon Myers, and Brandon Holt made possible our finest and most impactful publications, appearing at the prestigious UW CSE Potentially Computer Science Conference, along with other shenanigans.

I enjoyed formative interactions with UW undergraduates. Cody Schroeder and Kristian Lieberg provided an enjoyable summer lesson in advising undergraduate research. Working with my CSE 351 class in summer 2013 reminded me how much I enjoy teaching.

Outside the department, Tom and Fran were consistent adventure companions, great friends, and providers of balanced perspective on the relative importance of grad school and life. Trips with Nick opened up new modes of self-powered travel in the mountains.

Finally, the constant support of my family played an enormous role in bringing me to graduate school in the first place and encouraging me to succeed on my own terms.

[2] Technically, we have not yet established an upper bound.

Chapter 1

# Introduction

Software is used to solve inherently concurrent problems at all scales, from power grid control to finance and from vehicle control to medical devices. Building reliable software demands programming models and tools that support transparent reasoning and strong guarantees about the execution of programs. Shared-memory multithreading is a programming model for concurrency and parallelism that has seen heavy use in mainstream programming languages, despite suffering several pitfalls arising from the implicit sharing of memory. The rapid adoption of multicore processors in machines from servers to phones has increased the use of shared-memory multithreading while simultaneously exacerbating its problems. Threads can interact through shared memory in subtle, timing-dependent ways that lead to errors, such as *data races*, whose unpredictable outcomes make them difficult to find, understand, and eliminate.

Despite great strides over the past few decades in research to improve the reliability of shared-memory multithreaded programs, a number of factors have deterred solutions from reaching real-world success: problems of scale and precision have defeated some static program analysis tools; the inertia of mainstream languages or incompatibility with legacy code has slowed adoption of new programming models and language advances; high execution time overheads have discouraged the use of run-time error-detection tools. This dissertation focuses on challenges impeding widely-deployed, accurate, and automatic detection of data races in shared-memory multithreaded programs at run-time.

## 1.1 Data races are exceptional.

A *data race* in a shared-memory multithreaded program is a pair of memory accesses to the same shared-memory location by different threads, where at least one of the two accesses is a write and no synchronization orders the two accesses. The accesses may execute in either order, yielding unpredictable program state. Intuitively: a read racing with a write may return the location's old or newly written value; the final value stored in a location when two writes race may be either of the newly written values. Data races notoriously result in problematic and confusing errors. The silent, unpredictable resolution of data races hurts reproducibility when testing and debugging and can play a part in other more complex shared-memory errors such as violations of atomicity or determinism. Furthermore, while the memory consistency models of modern mainstream programming languages [20, 76] guarantee that programs free of data races execute with the intuitive semantics of *sequential consistency* [3, 69], they make weak or no guarantees about the execution of programs with data races. Optimizations in modern compilers and multiprocessor architectures can interact with data races to result in unintuitive, program executions where in-order interleavings of the operations of multiple threads are insufficient to reason about all possible executions.

Some researchers have proposed *data-race exceptions* to mitigate the ill effects of data races by making *every* data race an explicit fail-stop error at run-time [2, 27, 46, 73, 78]: on at least one access in a pair of racing accesses, raise an exception instead of executing the access. Under this model, data races become obvious at run-time and execution never reaches a racing access nor its ill effects. Early and explicit failure is arguably more useful to programmers than silently continuing execution. Data-race exceptions thus aid debugging and promote the treatment of data races as errors that must be fixed, rather than low-priority issues to ignore until obviously problematic. Furthermore, many analyses of higher-level properties of multithreaded programs, such as determinism [94] and atomicity checking [56], rely on data-race freedom or must perform data-race detection as part of the analysis. Data-race exceptions also simplify semantics in programming languages. In a semantics where data races are the only source of sequential consistency violations, raising an exception instead of allowing a data race suffices to avoid violations of sequential consistency.

## 1.2 Problem: Accurate language-level data-race detection is slow.

Implementing data-race exceptions naturally requires *accurate* dynamic data-race detection. An *accurate* dynamic data-race detector admits an execution (or a single access within that execution) if and only if it is data-race-free; it reports a data race if and only if a data race truly occurs in that execution. Much of the data-race detection literature uses the terms *sound* and *complete* to describe the two conditions for what we term *accuracy*, although application of these terms in the literature differs based on the perceived purpose of a data-race detector. This dissertation takes the positive purpose to *verify that executions are data-race-free*, and thus *admit executions if and only if they are data-race-free*, analogous to a type checker whose purpose is to admit only type-safe programs. A *sound* dynamic data-race detector admits an execution *only if* it is data-race-free. A *complete* dynamic data-race detector admits an execution *if* it is data-race-free.[1]

We use *data-race detection* to refer to *dynamic data-race detection* throughout this dissertation unless otherwise noted. We also assume *synchronous* (or *on-the-fly*) dynamic data-race detection, with data races reported immediately before one of the two racing accesses, as opposed to *asynchronous* or *post-mortem* data-race detectors which may report data races later or analyze a log to identify data-races after program execution is complete.

Implementing accurate dynamic data-race detection is challenging. Most techniques sacrifice guarantees of accuracy, potentially missing true data races or reporting false data races, or they incur heavy performance overhead. Previous hardware data-race detectors (*e.g.*, [83, 103, 143]) are inaccurate and unsuitable for data-race exceptions. While recent advances have made significant improvements to the performance of accurate dynamic software data-race detection, data-race exception support does not yet have performance suitable for production settings. FastTrack, the state of the art algorithm for pure-software dynamic data-race detection, is accurate on high-level programs, but may cause programs to run roughly an order of magnitude slower than under native execution [52, 55]. As is,

[1]Some other work aims to *find executions that contain data races*, where *sound* means reporting an execution only if it contains a data race and *complete* means reporting an execution if it contains a data race. We do not use these definitions.

neither hardware support nor pure-software dynamic data-race detection suffices to support accurate and fast data-race exceptions. This dissertation addresses three specific accuracy and performance problems that make data-race exceptions infeasible to date.

### 1.2.1 Fast hardware-supported data-race detection is inaccurate.

Hardware-supported data-race detectors are attractive for being fast and general, especially when applied to support data-race exceptions. Hardware can optimize common cases of an analysis to achieve much faster performance than pure software data-race detection implementations. Furthermore, by building in data-race detection at a low level in the system implementation stack, a hardware data-race detector can be reused across many software systems. The core performance- and correctness-critical components can be designed and implemented once and reused across systems. Unfortunately, previous work on hardware data-race detectors has produced only best-effort implementations whose accuracy is limited by fixed-size hardware resources or fundamentally inaccurate detection algorithms [73, 78, 82, 83, 103, 104, 143]. While many of these hardware-supported data-race detector designs have achieved very low performance overheads, none has achieved fast *and* fully accurate data-race detection prior to the work described in this dissertation. Fast, hardware-supported data-race exceptions have thus remained infeasible.

### 1.2.2 Low-level data-race detection is inaccurate on high-level languages.

Recent proposals for low-level dynamic data-race detection, including our work in Chapter 3 to address the problem of §1.2.1, have full accuracy for low-level programs and improved performance. An accurate *low-level data-race detector* analyzes virtual memory accesses in the instruction set architecture (ISA) and stores access history for virtual memory locations, reporting data races in this abstraction accurately. Low-level implementation allows for (1) hardware optimization of common cases and (2) reuse of fast data-race detection mechanisms by many software systems.

Naïvely, one might run a high-level language implementation like a Java virtual machine (JVM) on hardware with low-level data-race detection support to implement *language-level* *data-race exceptions* in the Java program. By *language-level data race*, we mean a data race between accesses in the high-level language memory abstraction. Unfortunately, neither low-level data races nor language-level data races subsumes the other. Thus a low-level race detector can report false data races and miss true data races for programs written in high-level languages.

Current hardware data-race detectors are designed to reason only about ISA-level programs and their data races. They cannot reason about data races in higher-level execution abstractions when non-trivial translation is involved. The mismatch between modern sophisticated language implementations and naïve hardware data-race detectors bars a large population of high-level language programs from the benefits of fast, accurate, and general hardware-supported data-race detection.

### 1.2.3 Software data-race detectors use costly defensive synchronization.

A dynamic data-race detector must protect its analysis metadata against data races in the analysis target. A software dynamic data-race detector stores its analysis metadata in shared memory. In general, the data-race detector may read and write metadata for memory location $x$ as part of a *barrier* immediately before every program access to location $x$. If two program accesses to $x$ are concurrent, even if they do not conflict (*i.e.*, even if both accesses are reads), metadata accesses will also be concurrent. The data-race detector must avoid ill effects of potential *metadata races* to ensure consistent and correct analysis results.

Pessimistic enforcement of analysis metadata consistency can be quite expensive. Profiling experiments in [38] suggest that synchronization to ensure metadata consistency accounts for 20%-90% of the overhead of the FastTrack [52] data-race detector on a suite of multithreaded benchmarks. The literature has generally assumed this problematic detail away as a mere artifact of implementation and not an interesting algorithmic feature. In fact, some real data-race detector implementations make no guarantee to avoid metadata races. When using data-race detection for debugging, this is often sufficient, as metadata consistency is rarely compromised in practice. This approach clearly does not suffice for data-race exceptions, where absolute accuracy guarantees are needed, but it does suggest an opportunity to exploit the rareness of problematic conflicts.

## 1.3 Dissertation Goals and Contributions

The goal of this dissertation is to design and evaluate techniques that reduce the overheads of accurate and fast data-race exceptions for high-level shared-memory multithreaded languages. Towards that goal, this dissertation offers three primary contributions in three areas of dynamic data-race detection: *race detection in software and hardware*, *low-level abstractable race detection*, and *fast instrumentation bias.*

### 1.3.1 RADISH: Accurate and Fast Race Detection in Software and Hardware (Chapter 3)

*Race Detection in Software and Hardware (RADISH)* is a hybrid software-hardware data-race detector that is both accurate on ISA-level programs and fast.

**Hypothesis:** A hybrid software-hardware system can exploit the accuracy of software data-race detectors and the performance of hardware support to build a hybrid data-race detector that is both accurate and fast. The flexible bounds of software implementation can maintain accuracy in all cases of analysis, while hardware support can reduce or eliminate latency for common cases.

#### Contributions

- We present the design of RADISH, a hybrid software-hardware data-race detector that retains the full accuracy of a software data-race detector while optimizing its common cases with fast hardware support. RADISH is the first hardware-supported data-race detector that is fully accurate on ISA-level programs.

- We show the accuracy of RADISH by its equivalence to a canonical vector-clock data-race detection algorithm.

### 1.3.2 LARD: Low-level Abstractable Race Detection (Chapter 4)

*Low-level Abstractable Race Detection (LARD)* virtualizes low-level dynamic data-race detection support to detect language-level data races in high-level programming languages.

**Hypothesis:** Hardware or other low-level implementations can enable fast and general support for accurate language-level data-race detection in high-level languages, yet current low-level solutions are accurate for ISA-level programs at best, and inaccurate for programs written in high-level languages.

**Contributions:**

- We present the first full explanation of how translation of a program from a high-level language to a low-level machine abstraction affects the primitives involved in the definition of a data race, and why low-level data-race detectors are incorrect for high-level programs as a result. We synthesize various issues encountered in earlier data-race detector implementations as well as issues not previously considered.

- We design *low-level abstractable race detection* (LARD), a simple interface for low-level data-race detectors and language implementations that virtualizes accurate low-level data-race detectors, allowing the construction of an accurate language-level data-race detector using a low-level data-race detector. We compare our approach to earlier systems that have addressed individual issues with low-level data-race detection.

- We implement our approach for Java, coupling a simulated hardware-supported ISA-level dynamic data-race detector and a modified Jikes RVM [8] Java virtual machine through a version of the x86 ISA extended with LARD primitives.

- We evaluate our implementation's accuracy, comparing against FastTrack [52], a naïve low-level data-race detector similar to our RADISH hybrid software-hardware data-race detector [38], and various partial implementations of LARD. We find that, in practice, naïve ISA-level data-race detectors suffer from false and missed data-races for Java programs, but LARD does not.

- We present cursory evaluation of our implementation's performance via simulation and via execution of the software component alone on real conventional hardware, finding that hardware support for language-level data-race detection is likely to provide good performance, similar to the performance of hardware-supported accurate ISA-level data-race detection.

### 1.3.3 Fib: Fast Instrumentation Bias for Pure-Software Data-Race Detection (Chapter 5)

*Fast Instrumentation Bias (Fib)* aims to reduce the cost of data-race checks in pure-software dynamic data-race detectors by using cooperative synchronization to protect metadata.

**Hypothesis:** Performance overheads of pure-software accurate dynamic data-race detection can be reduced by replacing pessimistic metadata synchronization with *instrumentation bias*, a form of cooperative metadata synchronization based on thread ownership information already implicitly encoded by data-race detection metadata.

**Contributions:**

- We present *Fast Instrumentation Bias (Fib)*, an algorithm for accurate data-race detection that avoids harmful metadata races with no synchronization in common cases, at the cost of expensive synchronization in rare cases.

- We apply conservative dynamic thread-escape analysis as a pre-filter for data-race detection and show how it maintains accuracy.

- We implement Fib for Java programs in the Jikes RVM [8] Java virtual machine.

- We evaluate the overall performance of multithreaded Java applications on our prototype implementation as well as unsynchronized and pessimistically synchronized implementations of FastTrack [52], finding that Fib is 13-21% faster than the fastest conventionally synchronized implementation on 4 benchmarks, 0-7% slower on three benchmarks, and 45-260% slower on three benchmarks. We profile the distribution of intended common and rare cases in practice, finding that poor performance in FIB is linked to relatively high rates of FIB's expensive slow paths.

- We discuss limitations of FIB as described in this dissertation, and propose future improvements.

## 1.4 Publication and Collaboration

Chapter 3 covers the design and accuracy of the RADISH data-race detector, published in the proceedings of the 39th International Symposium on Computer Architecture [38] with an accompanying technical report [39]. Joe Devietti led the RADISH project, in collaboration with this author, Karin Strauss, Luis Ceze, and Dan Grossman. Joe is responsible for a majority of the design and the full evaluation in [38]. Chapter 3 focuses on the accuracy guarantees of RADISH's optimizations, where the author's contributions were focused, and reproduces enough of the design of RADISH from [38] to support this discussion. System detail, simulation, and evaluation are covered in [38]. The correctness discussion in this chapter supersedes previously published versions.

Chapter 4 was published in the proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems [137]. The LARD project was led by the author, in collaboration with Dan Grossman and Luis Ceze. This dissertation makes minor changes to the original paper to consolidate some background and related work.

Chapter 5 covers unpublished work led by the author in collaboration with Man Cao, Mike Bond, and Dan Grossman. We plan to extend and publish this work in a future paper to supersede this presentation. Implementation of the base dynamic thread-escape analysis used in §5.4.1 and some related insights are due to Man and Mike. Specifically, the author's original proposal for accurately filtering data-race checks to non-escaped objects required the current epoch to be recorded as the last read and last write for all access histories in an object when that object is about to escape, which is sound and complete under the first-race guarantee. Man and Mike observed that it is also sound under the first-race guarantee to make no updates to an access history upon escape, as described in §5.4.1.1.

# Chapter 2

# Foundations and Related Work

In this chapter, we first present foundations of data races (§2.1) and techniques for accurate dynamic data race detection (§2.2). Then, we discuss other approaches to data-race detection that are not suited for data-race exceptions because they may report false data races (§2.3), miss true data races (§2.4), or both (§2.5), as well other issues in data-race detection. We briefly survey static techniques for data-race detection (§2.6). Previous work related specifically to Radish, Lard, and Fib is presented in §3.4, §4.5, and §5.7, respectively.

## 2.1 Data Races

We consider program execution traces ($T$) expressed as sequences of operations ($a, b$) of threads ($t, u$), using syntax shown in Figure 2.1. Reads ($\mathsf{rd}(t, x, v)$) and writes ($\mathsf{wr}(t, x, v)$) by a thread, $t$, load and store values, $v$, in memory locations, $x$. Synchronization operations, include thread $t$ forking a new thread $u$ ($\mathsf{fork}(t, u)$), thread $t$ joining an existing thread $u$ by blocking until thread $u$'s final operation has completed ($\mathsf{join}(t, u)$), and thread $t$ acquiring ($\mathsf{acq}(t, l)$) or releasing ($\mathsf{rel}(t, l)$) a lock.[1] We omit the standard formal semantics of the execution of traces and additional types of synchronization. Previous work [46, 52] covers these omissions using similar syntax.

[1]We assume sequential consistency. Although this breaks down in the presence of a data race, it does so only after the first race [5]. We discuss sequential consistency in §2.1.2, data-race detectors focused on sequential consistency violations in §2.4.1, and the first-race guarantee in §2.2.3.

Memory Location $x$ Lock $m$ Thread ID $t, u$ Value $v$

$$\begin{array}{rl} \text{Operation} & a, b ::= \mathsf{wr}(t, x, v) \mid \mathsf{rd}(t, x, v) \mid \mathsf{acq}(t, l) \mid \mathsf{rel}(t, l) \mid \mathsf{fork}(t, u) \mid \mathsf{join}(t, u) \\ \text{Trace} & T ::= \cdot \mid T, a \end{array}$$

**Figure 2.1:** Syntax of execution traces

### 2.1.1 The Happens-Before Relation and Data Races

The *happens-before* relation ($\xrightarrow{hb}_T$) is a strict partial order over operations in trace $T$, composed of the transitive closure over *program order* ($\xrightarrow{po}_T$), the order of operations within each thread in the, and *synchronization order* ($\xrightarrow{so}_T$), the ordering between synchronization operations in different threads [68]. Program order is straightforward: if operation $a$ of thread $t$ precedes operation $b$ of thread $t$ in trace $T$, then $a \xrightarrow{po}_T b$. Synchronization order is defined as follows:

- If thread $t$ forks thread $u$ at operation $a$ in $T$ and operation $b$ is the first operation of thread $u$ in $T$, then $a \xrightarrow{so}_T b$.[2]

- If operation $a$ is the last operation of thread $u$ in $T$ and thread $t$ joins thread $u$ at operation $b$ later in $T$, then $a \xrightarrow{so}_T b$.

- If thread $t$ releases lock $l$ at operation $a$ in $T$ and thread $u$ acquires lock $l$ at operation $b$ later in $T$, then $a \xrightarrow{so}_T b$.

A *data race* is a pair of *concurrent*, *conflicting* memory accesses [91]. Two distinct operations $a$ and $b$ are *concurrent* in trace $T$ if and only if they are not ordered by the happens-before relation (neither $a \xrightarrow{hb}_T b$ nor $b \xrightarrow{hb}_T a$). Two accesses *conflict* if they access the same

[2]We assume $b$ follows $a$ in $T$. The execution semantics of traces get stuck at operation $b$ if it does not follow operation $a$ in $T$.

location and at least one of the accesses is a write. More detailed characterizations of data races may be found in [88, 91].

### 2.1.2 Data Races and Higher-Level Properties of Program Executions

Data races are a fundamental type of error in shared-memory multithreaded programs. Data races are closely linked to several other properties of such programs. Relaxed memory consistency models for programming languages [20, 21, 76] and hardware [3–5] generally guarantee sequential consistency [69] given data-race-freedom, but in executions with data races, a weaker (or undefined) semantics applies. Thus programmers cannot reason about data races in terms of simple in-order interleavings of the operations of threads in general. Data races and data-race-freedom also factor in other higher-level properties (and their analyses) such as atomicity (*e.g.*, [56, 119]) and determinism (*e.g.*, [11, 37, 94]).

## 2.2 Accurate Dynamic Data-Race Detection

All accurate data-race detectors track the happens-before relation to check if pairs of conflicting accesses are concurrent, and thus racing.

### 2.2.1 Vector Clocks

This dissertation considers data-race detectors derived from a canonical algorithm using vector clocks [49, 80] to track the happens-before order induced by synchronization and recording a history of accesses to each location to determine if a current access is concurrent with any previous conflicting accesses. We present the base algorithm (formalized in more detail in [10]), followed by discussion of many variants in the literature.

A *vector clock*, v, contains an integer logical clock, $c$, for each thread, indexed by thread. Vector clocks represent frontiers in logical time. Under the formulation of the happens-before relation as a directed acyclic graph, vector clocks summarize information about a vertex representing an operation by a thread and this vertex's most recent predecessors from each other thread.

Figure 2.2 shows the canonical vector-clock data-race detection algorithm as a judgment

$$\begin{array}{rccl}
\text{Clock} & c & \in & \mathbb{N} \\
\text{Vector Clock} & \mathrm{v} & ::= & \cdot \mid \mathrm{v}, t \mapsto c \\
\text{Thread VCs} & C & ::= & \cdot \mid C, t \mapsto \mathrm{v} \\
\text{Lock VCs} & L & ::= & \cdot \mid L, m \mapsto \mathrm{v} \\
\text{Last Reads} & R & ::= & \cdot \mid R, x \mapsto \mathrm{v} \\
\text{Last Writes} & W & ::= & \cdot \mid W, x \mapsto \mathrm{v} \\
\text{Detector State} & (C; L; R; W) & &
\end{array}$$

**Figure 2.2:** Vector-clock data-race detector metadata

$(C; L; R; W) \overset{a}{\Longrightarrow}_{\mathsf{VC}} (C'; L'; R'; W')$ on the data-race detector state, shown in Figure 2.2, and a program operation. Program and heap constraints are imposed by an external judgment (not shown here) that uses this judgment for data-race detection over traces. Stuck-ness indicates a data race. We describe synchronization tracking in §2.2.1.1 and access tracking and checking in §2.2.1.2.

#### 2.2.1.1 Synchronization tracking

To track the happens-before order, the vector-clock data-race detector maintains:

- a vector clock, $C_t$, for each thread $t$, representing the last logical time in each thread that happens before the current logical time in this thread. The $t$th entry in thread $t$'s vector clock is its *local time*: $C_t(t)$.

- a vector clock, $L_l$, for each lock $l$, representing the last logical time in each thread that happens before the last release of the lock $l$.

The main thread's vector clock starts with a local time of 1 and all other vector clocks initially hold exclusively zero entries: $C_t(t) = 1 \wedge \forall u \neq t, C_t(u) = 0$. Vector clocks are updated on the execution of synchronization operations in a program trace as follows:

- VC FORK: When thread $t$ forks thread $u$, thread $u$'s vector clock is initialized with a copy of thread $t$'s current vector clock. Afterwards, both threads' local clocks are incremented.

$$\boxed{(C; L; R; W) \xRightarrow{a}_{\mathsf{VC}} (C'; L'; R'; W')}$$

**Synchronization tracking:**

VC FORK
$$\frac{\mathrm{v} = C_t, t \mapsto C_t(t) + 1 \qquad \mathrm{v}' = C_t, u \mapsto 1}{(C; L; R; W) \xRightarrow{\mathsf{fork}(t,u)}_{\mathsf{VC}} (C, t \mapsto \mathrm{v}, u \mapsto \mathrm{v}'; L; R; W)}$$

VC JOIN
$$\frac{\mathrm{v} = C_t \sqcup C_u}{(C; L; R; W) \xRightarrow{\mathsf{join}(t,u)}_{\mathsf{VC}} (C, t \mapsto \mathrm{v}; L; R; W)}$$

VC RELEASE
$$\frac{\mathrm{v} = C_t, t \mapsto C_t(t) + 1 \qquad \mathrm{v}' = C_t \sqcup L_m}{(C; L; R; W) \xRightarrow{\mathsf{rel}(t,l)}_{\mathsf{VC}} (C, t \mapsto \mathrm{v}; L, m \mapsto \mathrm{v}'; R; W)}$$

VC ACQUIRE
$$\frac{\mathrm{v} = C_t \sqcup L_m}{(C; L; R; W) \xRightarrow{\mathsf{acq}(t,l)}_{\mathsf{VC}} (C, t \mapsto \mathrm{v}; L; R; W)}$$

**Access checking:**

VC READ
$$\frac{W_x \sqsubseteq C_t \qquad \mathrm{v} = R_x, t \mapsto C_t(t)}{(C; L; R; W) \xRightarrow{\mathsf{rd}(t,x,v)}_{\mathsf{VC}} (C; L; R, x \mapsto \mathrm{v}; W)}$$

VC WRITE
$$\frac{W_x \sqsubseteq C_t \qquad R_x \sqsubseteq C_t \qquad \mathrm{v} = W_x, t \mapsto C_t(t)}{(C; L; R; W) \xRightarrow{\mathsf{wr}(t,x,v)}_{\mathsf{VC}} (C; L; R; W, x \mapsto \mathrm{v})}$$

**Figure 2.3:** Vector-clock data-race detector

This represents the fact that the fork operation, and all operations that happen before it, happen before all operations in thread $u$, but later operations of the two threads are concurrent with each other unless more synchronization is performed.

- VC JOIN: When thread $t$ joins on thread $u$, thread $t$'s vector clock is updated to be the entry-wise maximum ( $\sqcup$ ) of the current vector clocks of threads $t$ and $u$.

  This represents the fact that all operations in thread $u$ happen before all operations in thread $t$ at and after the join.

- VC RELEASE: When thread $t$ releases lock $l$, lock $l$'s vector clock is updated to the entry-wise maximum of the vector clocks of thread $t$ and lock $l$. (Note this maximum is always equivalent to thread $t$'s vector clock, since lock $l$'s vector clock was merged into thread $t$'s vector clock on the preceding acquire, and no other thread may have released the lock since.)

  This represents the fact that all operations that happen before the thread $t$'s release of lock $l$ will also happen before any later acquisitions of lock $l$.

- VC ACQUIRE: When thread $t$ acquires lock $l$, thread $t$'s vector clock is updated to be the pairwise maximum of the current vector clocks of thread $t$ and lock $l$.

  This represents the fact that all operations that happen before the last release of lock $l$ also happen before thread $t$'s acquisition of the lock and all of thread $t$'s later operations.

Other types of synchronization are handled similarly. *Outgoing* synchronization (*e.g.*, lock release) merges from the performing thread's vector clock into some other vector clock, representing the source of some happens-before edge that may later be completed when a thread performs incoming synchronization on the same target (analogous to a message sent in Lamport's formalism [68]), and then advances the thread's local time to show that later operations did not happen before this outgoing synchronization. *Incoming* synchronization (*e.g.*, lock acquire) merges some other vector clock into the performing thread's vector clock, representing the destination of a happens-before edge (analogous to a message received in Lamport's formalism).

#### 2.2.1.2 Access tracking and checking

For each memory location, an *access history* records the logical times of accesses to that location. The logical time of each subsequent access is checked against these previous accesses to determine whether they are conflicting and concurrent, thus racing. More specifically, for a location, $x$, the access history records:

- a set of *last reads*, $R_x$, encoded as a vector clock, where each thread's entry records the local time of that thread's last read access to the associated location.

- a set of *last writes*, $W_x$, encoded as a vector clock, where each thread's entry records the local time of that thread's last write access to the associated location.

Initially, all last writes and reads are set to 0. When a memory access is executed, it is first checked against the access history. If it does not race with previous accesses, it is recorded. The check and access history update for an access are assumed to happen atomically with the access itself. The checks are as follows:

- VC READ: On a *read* access, $\mathsf{rd}(t, x, v)$, to $x$ by thread $t$, for any thread $u \neq t$, if thread $t$ has not synchronized with thread $u$ since thread $u$'s last write to $x$ ($W_x(u) > C_t(u)$) then this access races with thread $u$'s last write to $x$. We express this in Figure 2.3 with the element-wise vector-clock happens-before operator $\sqsubseteq$. If no race is detected, then replace thread $t$'s last read for $x$ with thread $t$'s current local time and allow the access.

- VC WRITE: On a *write* access, $\mathsf{wr}(t, x, v)$, to $x$ by thread $t$, perform the same check as for the read case. Additionally, for any thread $u \neq t$, if thread $t$ has not synchronized with thread $u$ since thread $u$'s last read of $x$ ($R_x(u) > C_t(u)$) then this access races with thread $u$'s last read of $x$. We express these checks in Figure 2.3 with the element-wise vector-clock happens-before operator $\sqsubseteq$. If no race is detected, then replace thread $t$'s last write for $x$ with thread $t$'s current local time and allow the access.

#### 2.2.1.3 Variants

Similar algorithms have appeared in much work on data-race (or *access-anomaly*) detection [40, 41, 63, 93, 101, 111, 116].

An important optimization of the vector-clock algorithm for data-race detection is to store information about only a *single* last write instead of a vector clock recording information about the last write from each thread [40]. Write accesses to a location must be totally ordered by the happens-before relation in data-race-free executions, thus the set of last writes encoded by the vector clock is redundant. By exploiting the guarantee that accuracy survives until the first data race (or the assumption that an access that races with previous accesses is never recorded in an access history), it is safe to store only the globally most recent write (§2.2.3). Since the single recorded last write therefore must happen after all previous writes to be recorded, an access that happens after this last write also transitively happens after these previous writes. This optimization lowers the space requirements for last-writes storage from linear in the number of threads to constant, for a single last write. It also reduces the cost of a checks for read accesses from linear in the number of threads to constant, since read checks compare against only a single last write. The complexity of checks for writes remains linear in the number of threads due to the last reads vector clock, but reads are more common than writes.

TRaDe [33] optimizes the vector-clock data-race detection algorithm with accordion clocks [35], an alternative vector clock representation that elides unused entries. Our RADISH data-race detector, described in Chapter 3 and in [38], is an optimized hybrid software-hardware implementation of the vector-clock algorithm. FastTrack [52] also optimizes the vector-clock algorithm. It employs the single last-write optimization and a similar optimization that allows storage of a single last read in the common case when reads to a location by different threads happens-before ordered. Although such ordering is not guaranteed, it occurs, for example, whenever there exists a lock $l$ that is always held when accessing a memory location $x$. The use of such locking disciplines is common in practice. Storing a single last read instead of a vector clock of last reads when possible saves space and allows write checks to execute in constant time in the common case and time linear in the number of threads in the rare case. We discuss FastTrack further in §5.2.

### 2.2.2 Alternative Happens-Before Representations

Goldilocks [46] is an accurate dynamic data-race detector that uses an *extended lockset*[3] for each memory location. After an access, this set initially holds the thread responsible for that access. When a thread in the set performs outgoing synchronization on some target (*e.g.*, releases a lock), that target is added to the set. When a thread performs incoming synchronization on a target in the set (*e.g.*, acquires a lock in the set), that thread is added to the set. A thread is allowed to access the location if that thread is in the location's set. Goldilocks keeps separate sets for reads and writes, analogous to the last reads and last writes in the canonical algorithm above. These *extended locksets* encode the subset of the happens-before relation that is descended from the last access to the memory location they track—in other words, the nodes in the happens-before graph that are reachable from the last access. To support this analysis, which potentially requires updating all sets on every synchronization operation, the Goldilocks implementation does lazy processing of synchronization operations, maintaining a history of synchronization operations and only forcing the update of the set for a location upon a new access to that location.

### 2.2.3 First-Race Accuracy

The accurate algorithms for dynamic data-race detection described above are technically accurate for a given memory location only through the first data race on that location or the first race in the execution, a consideration discussed starting with earlier work on dynamic data-race detection and throughout the literature [10, 32, 52, 82, 90, 91].

The first-race rule is necessitated by the fact that none of these algorithms stores the complete history of accesses. Instead, they store a set of accesses with the invariant that all earlier accesses not in the set happen before at least one of the accesses in the set.[4]

[3]The meaning of Goldilocks' locksets is different than that of those in the lockset algorithm for data-race detection, discussed in §2.3.1.

[4]The actual invariant is stronger, as is its use: all earlier writes not represented in the write set happen before all writes represented in the write set and all reads represented in the the read set; all earlier reads not represented in the read set happen before at least one read represented in the read set and are not

Thus by the transitivity of happens-before, if all accesses in the set happen before some new access, then all earlier accesses happen before the new access. Real implementations, like the algorithms above, bound the size of the set based on the number threads (or a constant: 1). On data-race-free accesses, replacing an old access in the set with a new access that dominates it is safe: the invariant is maintained. However, replacing an access in the set with a racing access can break that invariant. Even though this first race has been reported, later racing accesses may appear to be data-race-free under the assumption that the invariant holds. An alternative for racing accesses is to report the race and let execution continue, but do not store the racing access. This clearly breaks the invariant as well, as the access is never recorded and future accesses that race with it will appear not to race. However, if the racing access is neither executed nor recorded, as with data-race exceptions, the invariant is maintained.

There are a few reasons why the *first-race* rule—and an even weaker guarantee to catch only the first race in the entire execution (not the first race per location in the execution)—is acceptable. For example, later data races may occur as a result of earlier data races. General dynamic data-race detection, the problem of finding all data races in an execution that are feasible independent of other data races in the execution, has been shown to be NP-hard [88, 91]. Choi and Min built a system to detect *race frontiers*, the first race in each process, in the context of reproducing data races [32]. In most relaxed memory models, the first race in an execution indicates the first possible end to the sequentially consistent execution prefix; reasoning beyond the first data race also requires consideration of non-sequentially consistent traces, not supported by most data-race detectors [5]. This is the foundation for a variation on data-race exceptions that raises an exception on all data races that violate sequential consistency (and potentially on more data races), but not necessarily on all data races [73, 78].

Most importantly for this dissertation, data-race exceptions make the first-race distinction moot: racing accesses are never actually executed, so there is never a first race or the accompanying compromise of accuracy, only races that would have been and accurate reports

concurrent with any write represented in the write set. Not all earlier reads are ordered with respect to all reads represented in the read set.

thereof. In fact, the first data race exception in each thread should correspond to Choi and Min's race frontiers [32].

### 2.2.4 Data-Race Exceptions

Data-race exceptions prevent data races and their effects from occurring by guaranteeing to raise an exception immediately before (*i.e.*, instead of) at least one of the two accesses in a data race. Typically, we think of raising an exception on exactly the second in time of the two accesses. However, allowing the exception to be raised on either or both of the accesses affords greater implementation flexibility and does not necessarily force serialization of concurrent accesses for analysis. Additional relaxations of the timing of exception delivery may offer more implementation alternatives. Regardless, we assume precise exception delivery.

Elmas, *et al.* [46], were the first to propose data-race exceptions and their benefits for mitigating the ill effects of data races. Other researchers have proposed or supported this idea and variants thereof [2, 27, 73, 78]. Specifically, exceptions for the subset of data races that induce sequential consistency violations are sufficient to simplify the memory consistency model in a way that avoids reasoning about non-sequentially consistent executions [73, 78]. A memory model with accurate data-race exceptions is stronger than the Java and C/C++ memory models, guaranteeing data-race freedom, or an exception, on each memory access. Data-race freedom still guarantees sequential consistency, and data-race-free programs never generate data-race exceptions. Full data-race exceptions provide this benefit, along with additional debugging benefits and support for other analyses that depend on data-race detection as a sub-analysis (*e.g.*, [56]) or require data-race freedom (*e.g.*, [94]). We have previously examined potential uses of recoverable data-race exceptions in designing new algorithms for synchronization or run-time systems [136].

### 2.2.5 Performance

Dynamic data-race detectors developed in industry have integrated lockset (§2.3.1) and happens-before algorithms, although they are generally heavyweight, running tens or hundreds of times slower than normal execution without data-race detection [61, 62, 84, 117, 125].

We do not discuss performance of inaccurate data-race detectors in depth as they are not suitable for our needs. Accurate data-race detectors have long had heavy performance overhead, but recent innovations have driven it down. Nonetheless, performance remains infeasible for always-on deployment.

The original Goldilocks implementation reports normalized run-times of 1-18× those of native execution, with most benchmarks under 5 or 6× in an interpreting JVM [46]. Using static data-race detection to elide checks helps reduce the overheads further. Results from [52] largely corroborate these results for the same implementation, but show that this implementation does have pathologically bad performance on some additional benchmarks (likely due to accommodations for its lazy updates of access histories). Slowdowns for an implementation of Goldilocks in the RoadRunner [54] dynamic analysis framework (via bytecode instrumentation on a JIT-compiling JVM) range from a few times to 77× [52]. An implementation of the DJIT+ algorithm [63] (an optimization of the vector-clock algorithm described in §2.2.1) in the same framework averages overheads of 20×, and FastTrack averages 8.5× [52]. Newer results for FastTrack show overheads averaging 7.5× alone and 5.7× after removal of redundant checks with RedCard [55].

Simulation results for the hardware-supported RADISH data-race detector show slowdowns between unnoticeable and as high as 3×, with most under about 2× [38]. Asynchronous checks (leading to imprecise exceptions) allow RADISH to use other cores and achieve overheads under about 2.25×, with most under 1.5×. Wester, *et al.*, use uniparallelism [127, 128] to parallelize the execution of data-race detection logic [135] in a way that allows the elision of defensive analysis synchronization, achieving small reductions in overhead for happens-before and lockset race detectors. Given more cores than the target program, their system can reduce data-race detection overheads roughly proportionately to the number of cores.

## 2.3 Conservative Data-Race Detection

Conservative data-race detectors never miss true data races, but do report false data races. Safety guarantees come at the expense of false warnings.

### 2.3.1 Lock Sets

The *lockset* algorithm for data-race detection assumes that data races are prevented by protecting every memory location with at least one lock that will be held by the accessing thread during *every* access to that location. Static [51], software dynamic [115], and hardware dynamic [143] analyses use lockset-based data-race detection. The basic algorithm checks the single-protecting-lock invariant by intersecting the sets of locks held at each access to a location. If this intersection ever becomes empty for a location, a data race is reported. This algorithm is sound—it never allows a racing access—but it is not complete—it disallows some non-racing accesses, because there exist well-synchronized programs that do not follow the single-protecting-lock pattern. On the other hand, lockset-based data-race detectors can report some data races that may be possible in other executions but did not occur in the observed execution. Not all lockset data-race reports indicate feasible data races, however. Generalization beyond the current execution can be useful in some settings, but for data-race exceptions, the desired semantics are to report exactly those data races present in the current execution.

RaceTrack [141] and MultiRace [101] use hybrids of happens-before and lockset to improve precision over lockset, though they do not achieve full accuracy.

### 2.3.2 Generalization to Other Executions

Other dynamic data-race detectors have refined the tendency of the lockset algorithm to generalize to other executions in more principled ways.

Smaragdakis, *et al.*, design an offline dynamic data-race detector that detects races with respect to *causal precedence*, a weaker ordering than happens-before that encodes constraints on whether operations *must* have happened in a particular order, as opposed to simply happening in said order due to scheduling decisions [121]. Their detector discovers causal-precedence races by analyzing a single concrete execution trace in polynomial time.

AccuLock [140] is a hybrid approach that uses happens-before reasoning for all non-lock synchronization and a modified lockset algorithm to track the effects of lock synchronization. Like a dynamic lockset data-race detector, it generalizes beyond the observed thread schedule to report data races possible in other thread schedules, but reports fewer infeasible data races than a pure lockset algorithm, due to its reasoning about other forms of synchronization besides locks.

## 2.4 Precise Data-Race Detection

Precise data-race detectors never report false data races, but may miss true data races. A number of dynamic data-race detectors employ optimizations that reduce performance overheads at the cost of missing some data races, while never reporting false data races. While these performance optimizations may be useful—and occasional missed races acceptable—in testing and debugging or in detecting a principled subset of data races, implementing data-race exceptions requires detecting all data races, rendering these techniques unsuitable.

### 2.4.1 Data Races that Violate Sequential Consistency

In the presence of memory access reorderings by the compiler or hardware, data races may let programs observe states of memory that correspond to no simple sequential interleaving of threads, violating *sequential consistency* [3, 4, 69]. The Java [76] and C/C++ [20] memory models both guarantee sequential consistency given data-race freedom, but if a data race occurs, a much weaker—-or undefined—semantics applies [2, 21]. Even if sequential consistency is preserved [73, 78], data races retain their unpredictable outcomes and can still contribute to other concurrency errors such as atomicity violations or determinism violations.

Under memory consistency models where data-race-freedom implies sequential consistency, violations of sequential consistency are possible when data races occur. Some data-race detectors have focused on detecting at least these sequential consistency-violating data races and raising an exception upon detection [73, 78, 120]. They provide a memory consistency semantics of *sequential consistency or exception*, which bans the least intuitive effects of data races. However, these systems do not detect all data races, which are still important for a number of other analyses and for debugging alone.

### 2.4.2 Other Precise Techniques

A number of other techniques for data-race detection are precise or complete (they allow all data-race-free accesses) but unsound (they allow some racing accesses).

Precise sampling dynamic data-race detectors, such as LiteRace [77], Pacer [22], SOS [72], and IFRit [45], perform data-race checks on a sample of accesses, rather than all accesses, with the goal of reporting some data races while reducing overheads. These sampling data-race detectors all guarantee not to report false data races, but they may miss true data races.

SOS [72] also optimistically identifies *stationary* memory locations, those locations that are not read-only in the language, but are observationally read-only in practice. For such fields, full data-race detection may be disabled, since read operations never race with one another. However, the algorithm for tracking stationary status is not fully sound, so some data races may be missed.

IFRit [45] uses an alternative algorithm for detecting data races that does not track the happens-before relation explicitly. An *interference-free region* is a region of a thread's execution including an access to a memory location and extending backwards to the most recent lock acquire (or other incoming synchronization) and forwards to the first lock release (or other outgoing synchronization) [43]. IFRit detects when interference-free regions for a single location in different threads overlap in real time. If at least one of these overlapping, conflicting regions includes a write to the location, then a data race has occurred. Since the regions overlap, the first outgoing synchronization after the access in one region cannot possibly have preceded the last incoming synchronization before the access in the other region.

Unsound dynamic thread-escape analysis, as used in Eraser [92, 115] and often used to filter analyses in the RoadRunner dynamic analysis framework [54], can lower the overheads of accurate data-race detectors by eliding full data-race detection analysis on accesses to memory locations that have thus far been accessed by only one thread, but it sacrifices soundness, losing the ability to determine whether the first conflicting access is concurrent or not. Since no access history is recorded in thread-local mode, it is too late to construct this history upon the first access by a second thread, so an analysis simply trusts that the first thread-escaping access does not race. We discuss conservative reachability-based dynamic thread-escape analysis as an accurate filter for data-race detection in §5.4.1.

## 2.5 Best-Effort Data-Race Detection and Other Tools

Model-checking may use an approximate abstraction of synchronization [60], leading to imprecision, or use the precise happens-before relation [64] when exploring feasible executions, but regardless, it will detect races only within the limited state-space it explores.

Some static [47], hybrid static-dynamic [30, 93, 105] and dynamic [48, 118, 141] data-race detectors are optimized for speed and heuristic usefulness of reports at the cost of some missed data races and false data races, via static heuristics [47], sampling or hardware performance counters and watchpoints [48, 118], and other approaches. Best-effort hybrid static-dynamic approaches are discussed further with static analyses in §2.6.

We consider data races at the granularity of atomic units of data, *e.g.*, fields in Java. However, some previous software race detectors uses object-granularity race detection as a less expensive approximation of true data-race detection [130]. General object-granularity race detection reports races on neither a superset nor a subset of accesses where an accurate data-race detector reports data races. Due to effects analogous to false sharing, object granularity tracking leads to both false data races and missed true data races. RaceTrack [141] is a dynamic data-race detector that begins with object-granularity detection and refines to field granularity once an object race is detected. OCTET [23], a framework to support certain concurrency analyses, uses object-granularity tracking of inter-thread dependences as a filter for client analysis checks, and is thus suited for analyses that tolerate this false sharing gracefully. We discuss OCTET further in §5.7. Hardware can make race detection or other analyses at the cache-line granularity much more efficient by observing cache coherence events or tracking state per cache line [38, 58, 82, 85, 110]. Data-race exceptions do not tolerate the inaccuracy of coarse-grained object or cache-line race detection. Finer-grained state must be maintained to maintain full accuracy (as in [38]).

Much work has focused on data-race detection for specific program structures such as nested fork-join programs (*e.g.*, [81]) and Cilk programs [28], event-based programs [109], parallel loop programs [7, 106], distributed memory systems or partitioned global address-space programs [63, 97, 98], as well as non-shared-memory message-passing environments [34, 89]. While synchronous online—or *on-the-fly*—data-race detection is required for data-race exceptions, several offline or *post-mortem* analysis techniques record information about a program execution and use it to detect data races after the fact [5, 31, 111], sometimes employing sampling [66, 118].

Early work focused on distinguishing independently feasible data races from data races dependent on others [88, 90]. More recent tools aim to distinguish data races that have observable ill effects from those that appear to be benign [65, 87], although such tools should make judgments only for particular compiled versions of programs. Legal compiler transformations may cause data races that seem benign at the source level—or that are indeed benign in a particular compilation of the program—to become harmful [19]. Dynamic analysis has been used to perturb worst-case legal behavior under language memory consistency models to expose unlikely but possible harmful effects of data races [53]. Other systems seek to survive the ill effects of data races via redundant execution [129], privatizing updates during critical sections [108], or letting programmers write after-the-fact *execution filters* to be applied to running programs [139].

## 2.6 Static Data-Race Detection

Restricted programming models [9, 15, 17, 57] or type systems [1, 24, 25, 51, 59, 79, 134] can statically prohibit programs with data races, completely avoiding the overheads of dynamic data-race detection, but they do so conservatively, prohibiting some valid data-race-free programs. Other static data-race detectors approximate possible run-time behaviors of general multithreaded programs with pointer analysis, symbolic execution, or other static abstractions of thread interactions [60, 86, 102, 124, 132]. In practice, an unsound escape hatch is usually provided to work around conservative analyses [57, 132]. These analyses are generally, but not always, conservative (*e.g.*, [132] may miss some races in rare cases).

When combined with dynamic analyses, conservative static analyses may prove some dynamic checks unnecessary, while the dynamic analysis can ensure precision even where the static analysis is conservative. Static thread-escape analysis has been used to elide unnecessary synchronization and can be used similarly to elide data-race checks on provably thread-local data [6, 29, 71, 112, 114]. Some data-race detectors have used more general static race detection analyses such as may-happen-in-parallel analysis as filters to elide dynamic checks of accesses [30, 93, 105, 131]. More recent work includes analysis to identify and remove provably redundant dynamic checks [55]. In theory, a hybrid static-dynamic approach supports fully accurate dynamic data-race detection with reduced run-time costs (as in [55]), but some hybrid static-dynamic analyses still sacrifice full accuracy (*e.g.*, [30, 105]).

Chapter 3

# RADISH: Accurate and Fast Race Detection in Software and Hardware

Chapter 3 covers the design and accuracy of the RADISH data-race detector, published in the proceedings of the 39th International Symposium on Computer Architecture [38] with an accompanying technical report [39]. Chapter 3 focuses on the accuracy guarantees of RADISH's optimizations, where the author's contributions were focused, and reproduces enough of the design of RADISH from [38] to support this discussion. System detail, simulation, and evaluation are covered in [38]. The correctness discussion in this chapter supersedes previously published versions.

## 3.1 Introduction

RADISH is a hybrid software-hardware data-race detector that is sound (missing no data races) and complete (reporting no false data races) with performance suitable for many deployment environments. RADISH is the first data-race detector to achieve this combination of accuracy and performance. RADISH optimizes the vector-clock algorithm for happens-before data-race detection (§2.2.1) by storing a useful subset of data-race detection metadata on-chip in a hardware-managed format. This on-chip metadata allows most data-race checks to occur completely in hardware with low latency. A simple software layer is responsible for persisting metadata when it overflows hardware resources, and for using this metadata to check for data races when hardware metadata is insufficient.

To help reduce the number of data-race checks that it must perform, RADISH uses cache coherence to detect when threads share data. To maintain accurate data-race detection at the byte level, RADISH augments coherence messages with per-byte access history information, but requires no changes to the actual coherence protocol. To keep hardware complexity modest, the only additions to each core are a small amount of state and logic for fast SIMD-style vector-clock computations. Crucially, and unlike many previous hardware proposals, the design of the timing-sensitive cache hierarchy is entirely unchanged by RADISH and there is no dedicated hardware storage for per-address data-race detection metadata. While on-chip and managed by hardware, per-address metadata is stored in dynamically allocated cache lines that share the cache data array with regular data. Thus there is no wasted hardware storage capacity when running programs that do not require dynamic data-race detection (*e.g.*, due to data-race-free programming models). Furthermore, RADISH can leverage type-safe languages to reduce overheads further.

The remainder of this chapter is organized as follows: §3.2 describes the RADISH algorithm along with its hardware and software components. §3.3 shows the equivalence of RADISH and the canonical software vector-clock data-race detection algorithm. §3.4 discusses related work and §3.5 concludes.

## 3.2 The RADISH System

This section discusses the intuition behind RADISH. Then, we describe what RADISH adds to a conventional processor design (§3.2.2), the metadata that RADISH uses (§3.2.3 and §3.2.4), the operations RADISH performs at each memory access (§3.2.5), and the RADISH software layer (§3.2.6). We conclude with a short example of RADISH's operation (§3.2.7).

### 3.2.1 Intuition

The intuition behind RADISH's hybrid hardware-software approach to data-race detection is to start with a software vector-clock data-race detector and map its most heavily used operations and metadata to hardware as frequently as possible. RADISH leverages three basic observations to accomplish this: (1) nearly all the work that a data-race detector must do occurs in response to coherence traffic, so the RADISH mechanisms are rarely activated outside these high-latency events; (2) the spatial and temporal locality exhibited by memory references extends to the metadata necessary for data-race detection, so the existing cache hierarchy can be used to accelerate metadata accesses; and (3) there is temporal locality in the data referenced by concurrently scheduled threads, so while RADISH caches metadata only for co-scheduled threads this is often sufficient to handle data-race checks completely in hardware.

We start from the vector-clock algorithm described in §2.2.1 and map parts of it into hardware structures as follows. Each core keeps a portion of its currently scheduled thread $t$'s local vector clock $C_t$ on chip, with an entry for each other actively executing thread. Thus, the size of this partial vector clock is bounded by the number of cores. When thread $t$ accesses byte $x$ and needs to access the metadata for $x$, $t$'s entries $R_x(t)$ and $W_x(t)$ in the last-reads and last-write vector clocks for $x$ are stored as metadata in the cache data array itself, using space otherwise available for data, but obviating the need for dedicated storage. Moreover, locations that can be proven data-race-free (*e.g.*, local variables) require no metadata, freeing cache capacity for other data or metadata. Lock vector clocks ($L_l$) are handled purely in software since they are accessed infrequently.

RADISH stores only the vector clocks for actively executing threads, and only a subset of read and write vector clocks for these threads, on the chip. Since this is only a subset of the full metadata needed for data-race detection on-chip, it is crucial to know when it is possible to reason soundly about data races from this partial view and when the full metadata is required. We solve this problem with three insights:

- We maintain *in-hardware status* information for each location $x$ that is cached on-chip, summarizing the number and type (read or write) of $x$'s last reads and last writes that are cached in hardware.

- We rely on software to virtualize limited hardware resources, by storing and providing access to metadata upon last-level cache evictions and context switches.

- We memoize the result of data-race checks using *local permissions*. This is particularly helpful for data-race checks performed in software, because they are expensive

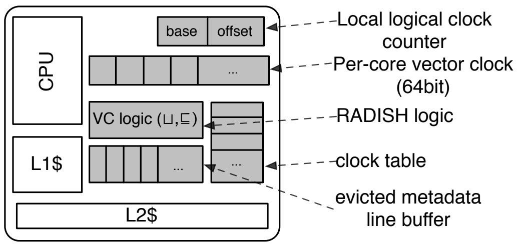

CPU
base
offset
...
VC logic (⊔,⊑)
L1$
L2$
Local logical clock counter
Per-core vector clock (64bit)
RADISH logic
clock table
evicted metadata line buffer

**Figure 3.1:** Overview of the RADISH processor core. State added by RADISH is shaded.

and require metadata not resident in hardware caches. Memoizing their results as permissions helps avoid repeated expensive checks.

The rest of this section explains in detail how RADISH implements these solutions.

### 3.2.2 The RADISH Architecture

RADISH makes only minimal changes to a conventional bus-based chip-multiprocessor architecture. Figure 3.1 shows the additional state added by RADISH with shaded blocks. A per-core vector clock contains a 64-bit clock for each processor in the system, including the local processor. The clock table manages the vector-clock values used by hardware; for efficiency, RADISH employs a reference-counting scheme that we describe in [38]. Finally, the RADISH logic implements the RADISH algorithm, including the vector-clock operations union ( $\sqcup$ ) and happens-before ($\sqsubseteq$), which can take advantage of SIMD parallelism. RADISH provides atomicity for each memory access, including its corresponding metadata access and data-race check, by detecting concurrent remote data accesses to the same location. Any such remote access must be the result of a data race. This mechanism uses RADISH's existing precise byte-level communication tracking.

Crucially, RADISH does not change the structure or timing of any portion of the cache hierarchy. Metadata is stored in the caches just like regular data is, and competes for cache capacity just like regular data does. This design choice ensures the critical path latency of cache hits is unchanged, and also ensures that the processor runs with full cache capacity when RADISH is disabled if data-race detection is not wanted for an application.

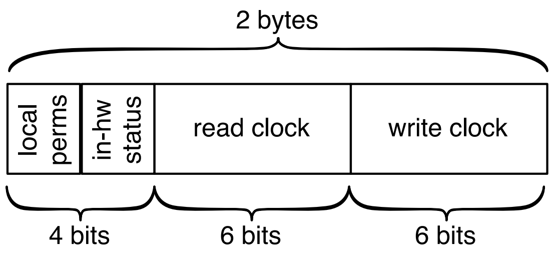


**Figure 3.2:** RADISH's in-cache metadata format. Local permissions have 3 possible values, and in-hardware status 4 values, so we use 4 bits to represent the 12 combinations. 6 bits are left for each clock; we discuss rollover issues in [38].

As metadata is allocated dynamically, any static information about data-race freedom can reduce RADISH's space and run-time overheads even further.

### 3.2.3 RADISH Metadata

RADISH maintains metadata for each location of virtual memory (discussed below), as well as the per-core partial vector clock mentioned previously. This per-core vector clock is accessible to software, as synchronization operations must read and write it. The vector clocks associated with synchronization objects are accessed relatively infrequently and thus can be managed by a RADISH-aware synchronization library.

RADISH uses hardware to cache a subset of the full metadata needed for data-race detection, and uses software to persist vector clock entries when they are evicted from the cache hierarchy. There can thus be two versions of any given entry of a vector clock—one in hardware and another in software. There are no version conflicts because hardware is always most up-to-date. §3.3 shows that RADISH's two vector clock versions can always be reconciled to the values a conventional vector clock data-race detector maintains.

RADISH maintains metadata for each byte in memory, since that is the finest granularity at which a program may access memory, according to modern memory models [20, 76], though RADISH can exploit type safety guarantees to soundly coarsen the metadata granularity for improved performance [38]. Without reliable information on data element size, tracking accesses with metadata for, *e.g.*, every 2 bytes could find false data races if two threads

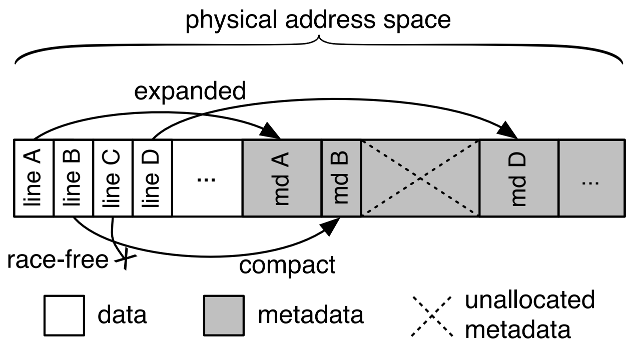


**Figure 3.3:** Mapping from data to metadata addresses for a single processor. Each processor uses a distinct portion of the physical address space.

concurrently write to the two different bytes.

The RADISH metadata for a byte of data consumes a total of 2 bytes of space (Figure 3.2), though a more compact 1:1 encoding is possible by leveraging type safety guarantees [38]. 2 bytes of metadata per byte of data admits a simple mapping from data to metadata: the location of the metadata for address $x$ is located at address $base + 2x$, where $base$ is chosen not to overlap with regular program data (Figure 3.3). Metadata addresses are special physical addresses that only ever reside in the cache tag arrays—hardware's metadata does not occupy any physical memory, as that would be redundant with software's representation. The metadata for a cache line's worth of data is split across two cache lines (as with cache lines A and D). These two metadata lines need not reside in the cache at the same time—metadata is fetched on demand based on the corresponding data being accessed. Other lines may use the compact metadata encoding (B) or may not need metadata at all (C) courtesy of a previous static analysis. The "holes" in the data-to-metadata mapping are unallocated and never fetched into the cache, making room for other useful (meta)data.

Each processor uses a distinct region of the physical address space for metadata. We can efficiently encode the fact that a line contains metadata, and also the processor $p$ to which the metadata belongs, by stealing some high-order bits from the physical address. This ownership information allows metadata to live in shared caches. Upon eviction from the last-level cache, metadata is stored by software, which uses its own opaque format that occupies virtual memory just like regular program data. Software-controlled metadata is never touched by hardware, so its format can be tuned to a particular run-time system, programming model, or even application, for maximum efficiency.

In addition to read and write clocks, the RADISH metadata consists of two additional pieces of state (Figure 3.2): in-hardware status and local permissions. This additional metadata is crucial for getting the most leverage from the metadata cached in hardware, so as to avoid consulting software in common cases.

#### 3.2.3.1 In-Hardware Status

*In-hardware status* encodes how much of the metadata for a given location resides in hardware. The status can be one of the following:

- EVERYTHING indicates that the metadata recording all last reads and the last write to the location is in cache.

- LASTWRITE indicates that the metadata recording the last write to a location is in cache.

- ALLLASTREADS indicates that the metadata recording all last reads of a location are in cache.

- INSOFTWARE indicates that software must be consulted to determine the most recent accesses for this location.

RADISH uses in-hardware status information to determine when a cache fill or data-race check, which would otherwise require consulting software, can in fact be done entirely in hardware.

#### 3.2.3.2 Local Permissions

*Local permissions* specify actions that the local thread can perform on a location that will definitely not result in a data race and thus do not require a data-race check. The allowed

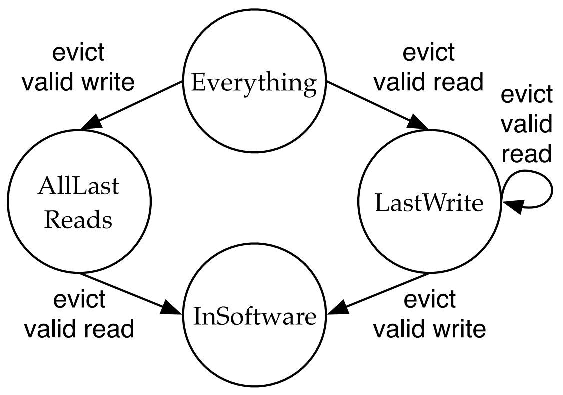


**Figure 3.4:** In-hardware status is downgraded when metadata is evicted from the last-level cache.

actions are encoded as *permissions*: a thread has WRITE, READ or NONE permissions to a location. WRITE > READ > NONE. Local permissions act as a filter, guaranteeing that no data-race check is necessary if local permissions allow an access. This allows RADISH to avoid performing a data-race check (whether in software or fully in hardware) on every memory access. Permissions violations may or may not be the result of an actual data race; additional work is required to disambiguate these cases.

### 3.2.4 Maintaining In-Hardware Status and Local Permissions

Metadata lines are not subject to the normal coherence protocol, but are instead updated on evictions of remote metadata lines, when local permissions are violated, and whenever coherence events occur on their associated data line. This latter property allows many metadata updates to piggy-back on regular coherence messages; metadata evictions and local permissions violations are the only sources of extra inter-processor communication in RADISH.

#### 3.2.4.1 Maintaining In-Hardware Status

On a data-race-free write, the **in-hardware status** for the bytes being written is set to EVERYTHING, because all earlier accesses happen before this write, so future checks need only consider the new last write. The Everything status propagates via coherence messages to subsequent reads and writes of $x$, and is only demoted when metadata for one of these reads or writes to $x$ is evicted, as detailed in Figure 3.4. Note a valid write clock is never evicted while the status is AllLastReads: since the valid last write $a$ was already evicted, another write $a'$ would need to occur, but $a'$ would have reset the status back to Everything. The in-hardware status can also be set on reads, if the read triggers a software vector-clock check that loads new metadata into hardware. If the check reveals that thread $t$ is the sole last reader of a location $x$ since the last write, then $t$'s read means the in-hardware status can now be set to AllLastReads (if the state was InSoftware) or Everything (if the state was LastWrite). In Radish, metadata evictions from the last-level cache require a broadcast to downgrade in-hardware status appropriately, but this cost can be masked by the L2 miss that triggered the last-level cache eviction.

#### 3.2.4.2 Maintaining Local Permissions

Local permissions are set for each byte $x$ in a line when it is brought into $t$'s cache, by broadcasting (concurrently with the fill) to gather the last accesses to $x$ by other scheduled threads. Thread $t$ then checks its happens-before relation with these remote accesses. Thread $t$ gets Write permission for $x$ if $t$'s current logical time happens after the last write to $x$ and all last reads of $x$ (*i.e.*, $W_x \sqsubseteq C_t$ and $R_x \sqsubseteq C_t$), Read if $t$'s current logical time happens after the last write to $x$ but not all last reads of $x$ (*i.e.*, $W_x \sqsubseteq C_t$ and $R_x \not\sqsubseteq C_t$), and None otherwise. In-hardware status determines whether the on-chip metadata suffices to attempt each of these checks. If byte $x$ has insufficient in-hardware metadata, Radish sets None permission for $x$; this is conservative but cheaper than consulting software to get a precise result, as $t$ may never access $x$.

When a remote thread $u$ performs a read or write of $x$, we need to update $t$'s local permission for $x$ to maintain its guarantee. If $u$ does a write to $x$, we must downgrade $t$'s permission on $x$ to None. If $u$ does a read, then $t$'s permission must be downgraded to Read as well, since $t$'s current logical time no longer happens after all last accesses to $x$, making a write unsafe. If $t$'s permission for $x$ was previously None, it is unchanged—downgrading never increases permissions. It is sound to perform downgrades only on coherence events

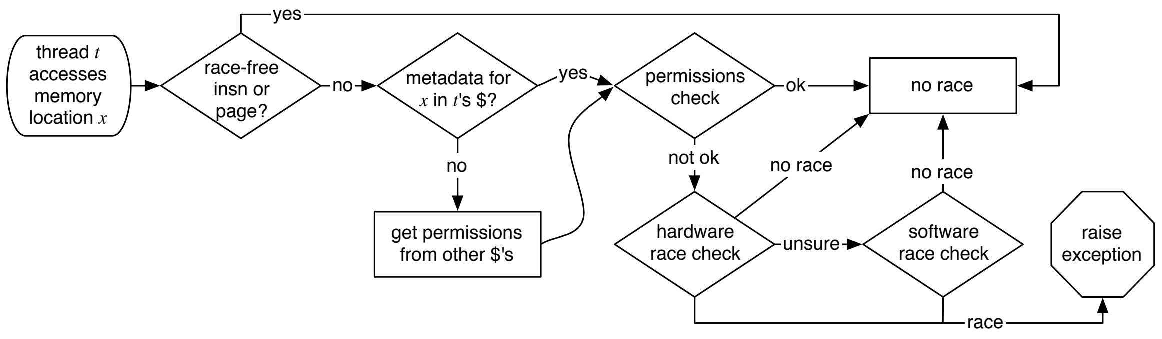


**Figure 3.5:** Flowchart describing when and how RADISH performs data-race checks for each memory access.

and cache fills as shown in §3.3.6.

### 3.2.5 RADISH Checks

Data-race checks in RADISH are a 3-stage process: a permissions check, a hardware data-race check, and a software data-race check (Figure 3.5). First, a thread $t$ accesses a memory location $x$. The load or store instruction may be marked statically as data-race-free (*e.g.*, a compiler may tag accesses to non-escaping local variables), or the location accessed may reside on a page tagged as containing only data-race-free locations (*e.g.*, thread-local storage); in these cases no further work is necessary. Otherwise, we consult $t$'s metadata for $x$; if the metadata is not in the local processor's cache it is fetched from other caches or software (§3.2.6).

Once thread $t$'s metadata for location $x$ is in cache, the first step of a RADISH data-race check is a **permissions check**: if local permissions allow the access, no further data-race checking is necessary. The last read or last write clock for $t$ must be updated if it is older than $t$'s current logical clock, but this update can happen locally. A **hardware data-race check** is performed if the permissions do not allow the access to proceed. A hardware data-race check consults the precise read/write clock values from other caches, together with the local in-hardware status metadata, to determine if the access to $x$ is data-race-free. Specifically, a read operation $a$ is data-race-free if (1) $a$ happens after some last-read operation $b$ (since $b$ must happen after the last write to $x$ or a data race would have been detected on one of those previous accesses) or (2) the in-hardware status for $x$ is LastWrite and $a$ happens after the last write. A write operation $w$ is data-race-free if the in-hardware status for $x$ is Everything and $w$ happens after the last write and all last reads.

Depending on the amount of state available in hardware, the outcome of the hardware data-race check may be that (1) there definitely is a data race, (2) there definitely is not a data race, or (3) there might be a data race, *e.g.*, there is no data race with respect to the in-cache data, but the in-hardware status is InSoftware so there is additional relevant information in software. A **software data-race check** is required only in case (3). To exploit spatial locality and amortize the overhead of invoking the software handler, the software check preemptively sets local permissions for all metadata in the metadata line $\ell$ that contains the metadata for $x$.

### 3.2.6 The Radish Software Interface

For Radish, the system's **synchronization library** must be modified to update the per-core vector clocks on synchronization operations, and to maintain a vector clock with each synchronization object. There are also software handlers for **data-race checks** and **metadata evictions**. The data-race check handler is called on a memory operation when hardware does not have enough information to prove data-race freedom. The data-race check handler may be called synchronously or asynchronously. The eviction handler is called when metadata is displaced and may be executed asynchronously with respect to memory operations. Thus, these handlers may execute on any available processor.

A **software data-race check handler** is passed, via registers, the physical address $p_x$ of the location $x$ that triggered the check, as well as the current hardware entries for the read/write vector clocks of $x$. The physical address $p_x$ is used to index the *variable map*: the central software data structure used by the Radish software layer. The variable map contains a mapping from physical addresses (of data) to software metadata (*e.g.*, read/write vector-clock pairs). Physical addresses are used because they are convenient identifiers—since caches are physically tagged, when evicting a metadata cache line we identify it with its physical address to avoid any need for reverse translation. These physical addresses do not allow the program unmediated access to physical memory. When the OS migrates physical pages, the variable map must be updated accordingly, but a physically-indexed map keeps the common case fast.

A software data-race check merges the hardware read/write vector clock entries with the values from software (obtained from the variable map) to obtain up-to-date read/write vector clocks ($R_x$ and $W_x$ in §2.2.1). Similarly, the entries from the per-core partial vector clock (accessible via new instructions) are used to obtain the up-to-date thread's vector clock ($C_t$). Once hardware and software values are merged, the standard happens-before data-race check occurs. A signal may be raised to indicate that a data race has occurred.

To amortize the cost of invoking software, software data-race checks verify the data-race freedom of a single memory access, but update hardware metadata for all locations in the cache line with metadata from software and set the in-hardware status for each location where software information is imported into hardware.

The **eviction handler** is called whenever a metadata line is evicted from the last-level cache. This handler is invoked with the physical address $p$ of the data corresponding to the metadata being evicted; $p$ is used to index the variable map described previously. To process metadata evictions asynchronously, evicted metadata lines are placed into a hardware buffer. The eviction handler reads from this buffer via special load instructions and updates the variable map with the hardware values.

In Radish, the software metadata representation is opaque to hardware, allowing software to freely optimize its metadata representation, *e.g.*, to save space [35, 52] or to leverage structured parallelism [81]. A software-managed metadata format also admits compatibility with compacting garbage collectors that move objects in memory. (See also related issues in §4.2.4.)

On a **context switch**, per-core partial vector clocks must be flushed into software, and all per-core clocks need to be updated to replace entries for the descheduled thread with entries for the incoming thread. The evicted metadata line buffer can be cleared lazily, as entries are tagged with their owner thread. The crucial issue is dealing with metadata lines belonging to the descheduled thread, as they can occupy several MB of state. We observe that software can conservatively approximate when its metadata is stale (*i.e.*, when there exists metadata in hardware that is more up-to-date) by setting a "potentially stale" bit whenever a software check occurs (as a check must occur on the very first access to a location, which brings metadata into hardware), and clearing this bit when sufficient metadata has been evicted.

Using software's conservative approximation of hardware state, metadata lines can be flushed eagerly or lazily. The eager approach uses software to flush these lines immediately at the context switch, bringing all software metadata up-to-date. Alternatively, flushing can be done lazily. At the cost of stealing extra unused bits from the physical address space, each thread ID can be allocated a region within the metadata space of a processor. Thread IDs can be reused across processes: as metadata lives in the physical address space, the same thread ID in two different processes will be isolated if the processes are isolated.[1] When asked to perform a data-race check, software determines if its metadata is potentially stale, and flushes a superset of the necessary entries (even for descheduled threads) from hardware caches.

### 3.2.7 An Example Trace

We now show Radish's operation on a short trace of instructions. Table 3.1 shows how hardware metadata (local permissions, read/write vector clock entries, per-core vector clocks, and in-hardware status) and software metadata (only read/write vector clocks are shown for simplicity) are updated in response to various events. For simplicity, events for a single location $x$ are shown. There are three threads, but only threads 0 and 1 are scheduled (on processors `p0` and `p1`, respectively).

Initially, there is no metadata cached in hardware. `p0`'s load triggers a software data-race check, because the in-hardware status InSoftware indicates that there is insufficient information in hardware to check the access. The software check brings information about

[1]If processes share memory, then shared regions must be flushed whenever a new process is scheduled. The physical address space could be shrunk further to accommodate process IDs, but the overhead (combined with thread IDs) is likely to be too great.

| Event | Software | | p0 running thread 0 | | | | | p1 running thread 1 | | | | | In-hardware |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reads | writes | MSI | perm | read | write | VC | MSI | perm | read | write | VC | status |
| (initial) | 7,0,0 | 2,0,0 | | | | | $\underline{9}$,4 | | | | | 6,$\underline{5}$ | InSoftware |
| p0 read | | | S | W | 9 | 2 | | | | | | | Everything |
| p1 read | | | | W | | | | S | R | 5 | 0 | | |
| p0 release | | | | | | | $\underline{10}$,4 | | | | | | |
| p1 acquire | | | | | | | | | | | | 9,$\underline{5}$ | |
| p1 write | | | I | N | 0 | 0 | | M | W | 0 | 5 | | |
| p0 evict md | | | - | | | | | | | | | | |
| p1 evict md | 0,0,0 | 0,5,0 | | | | | | - | | | | | InSoftware |

**Table 3.1:** An example trace showing how Radish metadata is updated. Empty cells indicate the value is the same as in the cell above. The local component of each core's vector clock is underlined.

p0's previous write at time 2 into hardware; combined with p0's read there is now complete information about $x$ in hardware and the in-hardware status is upgraded to Everything. Furthermore, p0 obtains Write permissions on $x$ because p0 can soundly read or write $x$ without the need for further data-race checks. Next, p1 performs a read that, due to the current in-hardware status, can be checked without consulting software. Note that p1 gets Read permissions, as it can perform further reads of $x$ without racing, but p0 retains its Write permissions (though an immediate write by p0 would race). However, since p0 has the cache line containing $x$ in Shared state, it cannot actually write to $x$ without triggering a coherence message. This message will downgrade p0's local permissions to Read, so the permissions check will fail and trigger a deeper check that will catch the data-race.

Next, synchronization occurs from p0 to p1, which updates both processors' vector clocks. Then p1 performs a write, which is well-synchronized with both the last write (by p0) and the last read (also by p0). p1's write downgrades the local permissions for p0, and clears all other hardware read/write clocks. Since p0's clocks have been reset, the read/write vector clocks in software do not change when p0 evicts the metadata for $x$. Software's metadata is stale, but is brought up-to-date with p1's eviction.

## 3.3 Equivalence to Canonical Vector-Clock Data-Race Detector

In this section, we show the soundness and completeness of RADISH via its equivalence to the canonical accurate vector-clock data-race detection algorithm (§2.2.1). We first show how RADISH's hybrid hardware and software metadata is equivalent to the metadata stored by a vector-clock data-race detector (§3.3.1). Next, we show how RADISH preserves this equivalence at each step in execution and thus reports the same data races, starting with a simple version of RADISH with no cache evictions (unbounded caches), no context switches, and no optimizations (§3.3.2), and progressively adding cache evictions (§3.3.3), context switches (§3.3.4), in-hardware status (§3.3.5), and local permissions (§3.3.6), showing that each maintains equivalence.

### 3.3.1 State

RADISH stores the full vector-clock data-race detector state $(C, L, R, W)$ (as introduced in §2.2.1) as a hybrid of hardware metadata $(\mathbb{C}, \mathbb{R}, \mathbb{W})$ and software metadata $(\mathcal{C}, \mathcal{L}, \mathcal{R}, \mathcal{W})$. We use the following notation throughout:

$\mathbb{C}_t(u)$ is thread $u$'s entry in thread $t$'s on-chip vector clock.

$\mathbb{R}_x(t)$ and $\mathbb{W}_x(t)$ are byte $x$'s last-read and last-write clocks in thread $t$'s cached metadata.

$\mathcal{C}_t(u)$ is thread $u$'s entry in thread $t$'s software vector clock.

$\mathcal{L}_l(t)$ is thread $t$'s entry in lock $l$'s software vector clock.

$\mathcal{R}_x(t)$ and $\mathcal{W}_x(t)$ are thread $t$'s entries in byte $x$'s last-reads and last-write software vector clocks.

RADISH's hybrid state (equivalent to the full vector-clock data-race detector state) is the software state, with elements overridden by hardware state where available as follows:

$$\begin{aligned} C_t &= \text{if } t \in \mathsf{Dom}(\mathbb{C}) \text{ then } \mathbb{C}_t \text{ else } \mathcal{C}_t \\ L_l &= \mathcal{L}_l \\ R_x(t) &= \text{if } t \in \mathsf{Dom}(\mathbb{R}_x) \text{ then } \mathbb{R}_x(t) \text{ else } \mathcal{R}_x(t) \\ W_x(t) &= \text{if } t \in \mathsf{Dom}(\mathbb{W}_x) \text{ then } \mathbb{W}_x(t) \text{ else } \mathcal{W}_x(t) \end{aligned}$$

The initial states of the vector-clock data-race detector and RADISH are equivalent under this mapping: RADISH has no metadata in hardware.

### 3.3.2 No Cache Evictions, No Context Switches, No Optimizations

We first consider a simple RADISH machine with unbounded caches (and hence no evictions of hardware metadata), no context switches, and no in-hardware status or local permissions.

#### 3.3.2.1 Synchronization Tracking

When a thread $t$ is forked and scheduled onto a core, 0 entries are stored in $\mathbb{C}_t$ for all running threads and $\mathbb{C}_t(t)$ is set to 1. A 0 entry for thread $t$ is stored in the hardware vector clock of each running thread $u$: $\forall$ scheduled $u, \mathbb{C}_u(t) = \mathcal{C}_u(t)$.

When synchronization tracking updates thread vector clocks, it updates entries in $\mathbb{C}$. Since all running threads remain scheduled onto a core and each core's vector clock $\mathbb{C}_t$ has space for entries of all cores, $\mathbb{C}$ contains the full vector clocks for all threads that have ever run in this execution. Entries in $\mathcal{C}$ remain at their initial values. Lock vector clocks are managed purely in software. Other operations have no effect on thread or lock vector clocks. Thus, under the derivation above, the RADISH thread ($\mathbb{C}$) and lock ($\mathcal{L}$) vector clocks remain equivalent to those of the canonical detector ($C$, $L$) through the execution.

#### 3.3.2.2 Access Tracking and Checking

On an access to address $x$ by thread $t$, metadata for address $x$ is loaded into cache from software if it is not already in cache. The hardware last-read and last-write clocks $\mathbb{R}_x(t)$ and $\mathbb{W}_x(t)$ are initialized from $t$'s entries $\mathcal{R}_x(t)$ and $\mathcal{W}_x(t)$ in the software last-reads and last-write vector clocks. Once metadata is in hardware, it overrides software metadata. Stale software metadata is never observed. Without cache evictions, we never need to persist hardware metadata back to software metadata, so software metadata remains empty.

On every memory access to $x$ by $t$, we perform the $\mathbb{W}_x \sqsubseteq \mathbb{C}_t$ hardware check and, for writes, the $\mathbb{R}_x \sqsubseteq \mathbb{C}_t$ hardware check from the vector-clock data-race detector. Without context switches or cache evictions, we are guaranteed that all non-zero last-reads and last-write metadata are in hardware, since only running threads may have previously accessed any memory locations and the last reads and last writes have not been evicted from cache. Thus these hardware checks, identical to the checks in the canonical algorithm, operate over equivalent metadata and therefore return equivalent results.

On a data-race-free write by thread $t$, we set the hardware metadata clocks $\mathbb{R}_x(u)$ and $\mathbb{W}_x(u)$ to zero for all threads $u$ that have $x$'s metadata in cache. For those threads $u$ that do not have $x$'s metadata in cache, the entries in the software vector clocks $\mathcal{R}_x(u)$ and $\mathcal{W}_x(u)$ are all ready zero, since no metadata is ever persisted back from hardware to software in this version, so we do not zero them explicitly. This is equivalent to zeroing all entries in full vector clocks $R_x$ and $W_x$, since hardware metadata always overrides software metadata when deriving the full state. We set the hardware metadata $\mathbb{R}_x(t)$ or $\mathbb{W}_x(t)$ when thread $t$ does a data-race-free read or write, respectively, which is equivalent to setting $R_x(t)$ or $W_x(t)$, since $t$'s metadata for $x$ is in cache and overrides any software metadata in $\mathcal{R}_x(t)$ or $\mathcal{W}_x(t)$.

Other operations (*e.g.*, synchronization) have no effect on access history metadata. Thus RADISH and the canonical algorithm perform equivalent data-race checks and updates on equivalent access history state, preserving the equivalence of access history state through the execution.

### 3.3.3 Cache Evictions

With the possibility of cache evictions, we cannot rely on all non-zero metadata being permanently resident in hardware. Hardware metadata entries $\mathbb{R}_x(t)$ and $\mathbb{W}_x(t)$ must be stored back to software vector clock entries $\mathcal{R}_x(t)$ and $\mathcal{W}_x(t)$ when evicted from cache. Data-race checks must run over the combined hardware-software state. Metadata updates are still made directly in hardware alone. Synchronization tracking is unaffected by cache evictions.

Data-race checks now use the full state derivation, using software access history metadata with any existing hardware metadata overriding the corresponding software entries, so we must show that hardware metadata entries are never older than the the software entries they override. Since software access history metadata is updated from hardware metadata only upon that hardware metadata's eviction and updates to hardware metadata discard older last writes or last reads only when they happen before the newer recorded accesses, this holds. Likewise, since hardware metadata is always persisted to software on eviction, no updates to hardware metadata are ever lost at eviction. It is straightforward to see how the combined RADISH state still remains equivalent to the canonical algorithm's state across accesses and metadata cache evictions.

### 3.3.4 Context Switches

Context switches require flushing RADISH's hardware metadata somewhat like a TLB shootdown, although unlike TLB entries, RADISH's in-hardware metadata can be newer than its software metadata, so the hardware metadata must be persisted back to the software layer in the general case. See §3.2.6 and [38] for additional discussion. RADISH's behavior for eagerly handled context switches is a special case of that for cache eviction, as discussed in §3.3.3, so it clearly maintains equivalence. Lazily handled context switches tag metadata in lower-level caches and persist them to software later, on eviction, examining the thread tag to assign them to the proper owners in the software representation. Metadata lookups in lower-level caches likewise examine this tag to properly distinguish metadata of different threads that have been scheduled to the same processor.

### 3.3.5 In-Hardware Status

In order to allow some data-race checks to skip examining software metadata without also missing some data races, in-hardware status (§3.2.3.1 and §3.2.4.1) must allow a pure-hardware data-race check only if a combined hardware-software check would not detect a data race.

Assuming in-hardware status does not overstate what parts of an address's metadata are fully in-hardware (by the definitions in §3.2.3.1), it serves as a safe filter for software data-race checks. When checking a read, ordering with the last write must be established. If the in-hardware status for $x$ is LastWrite or Everything, the read check may proceed in hardware without loading additional metadata from software. When checking a write, ordering with the last write and all last reads must be established. If the in-hardware status for $x$ is Everything the the check may proceed without loading additional metadata from software. If the in-hardware status in either case does not suffice, the full metadata $W_x$ and (for writes) $R_x$ must be assembled from what is in hardware and what is in software.

It is straightforward to see that in-hardware status is downgraded on evictions of metadata and upgraded on successful checks or cache fills as detailed in §3.2.4.1 and Figure 3.4. Whenever metadata containing a last read or last write of address $x$ is evicted, $x$'s in-hardware status is downgraded so future checks on address $x$ see that either the last write or some last reads are not present in hardware. Checks loading metadata from software or changing the set of last writes/reads may upgrade the status if they introduce (or eliminate) missing pieces of metadata in hardware.

### 3.3.6 Local Permissions

Local permissions must preserve two invariants: If local permissions allow an access then:

**Local checks suffice:** The access cannot result in a data race with an earlier or simultaneous access. In other words, if local permissions allow an access then a full data-race check on that access would succeed. If other threads' permissions are downgraded accordingly whenever one thread performs a memory operation, this invariant holds. It is also sufficient to downgrade permissions only on coherence events, which we demonstrate here.

**Local updates suffice:** Necessary metadata updates are limited to the executing thread's last read and last write. In other words, no metadata updates are required outside the metadata present in the checking thread's cache.

These two invariants together guarantee that each access check and update resolved by local permissions is equivalent to the check and update of the full data-race check.

#### 3.3.6.1 Proof: Local Checks Suffice for Permitted Accesses

We characterize a coherence event on data line $\ell$ initiated because of a memory access $a$ by thread $t$ in a program trace $T$ as a set $e(\ell) = \{a\} \cup \{e_u(\ell) \mid u \neq t\}$ of events in each other thread that occur atomically (and thus consecutively) immediately preceding $a$ in $T$. (Operations $a$ in traces $T$ have implicit sequencing identifiers to distinguish syntactically identical but dynamically distinct operations. We elide them to reduce clutter.) For each thread $u$, $e_u(\ell)$ happens after all events $a$ by thread $u$ that precede $e_u(\ell)$ in $T$ and $e_u(\ell)$ happens before all events $b$ by thread $u$ that follow $e_u(\ell)$ in $T$. We attribute an update of $u$'s permissions for an address $x$ during $e(\ell)$ to $e_u(\ell)$ if $u \neq t$, or to $a$ for thread $t$.

Let $\ell$ be a data line containing $x$, let $b$ be a memory operation on $x$ by thread $t$ in program trace $T$, let $d$ be the last event that precedes—or is—$b$ in $T$ on which $t$'s local permission for $x$ is set, and let $a \neq b$ be an access to $x$ that conflicts with $b$ (at least one of $a$ or $b$ is a write) and precedes $b$ in $T$. It suffices to show that if $d$ sets permissions that allow $b$ then $a$ must happen before $b$.

The proof is by contradiction. Suppose $d$ sets permission $p$ for $t$ on $x$ that allows $b$, where $a$ does not happen before $b$. Then $p$ must be READ or WRITE to allow $b$. For all vector clocks $v \in \{C_t, R_x, W_x\}$, let $v^f$ denote their values immediately preceding event $f$.

If $d$ is a memory operation, then it performs a full check showing $W_x^d \sqsubseteq C_t^d$ and, if $d$ sets WRITE, that $R_x^d \sqsubseteq C_t^d$. If $d = b$, then clearly $a$ must happen before $b$ since it is represented in $W_x^d$ or $R_x^d$. This is a contradiction. Otherwise, by program order, $d$ happens before $b$, and $W_x^d \sqsubseteq C_t^b$ in both cases and $R_x^d \sqsubseteq C_t^b$ in the WRITE case. Since $a$ does not happen before $b$, then $a$ must follow $d$ in $T$. Also, at $d$, $t$ must have $\ell$ in non-invalid state. If $a = \mathsf{wr}(u, x, \beta)$ then there must be some coherence event $e(\ell)$ in $T$ between $d$ and $a$ to give $u$ line $\ell$ in modified state, where $e_t(\ell)$ would downgrade $t$'s permission on $x$ to NONE, so $d$ is not the last event to set $t$'s permission on $x$ before $b$, which is a contradiction. Otherwise, $a = \mathsf{rd}(u, x, \beta)$, $b = \mathsf{wr}(t, x, \beta)$, and $d$ must set WRITE if $b$ is to be allowed. Then there must be some coherence event $e(\ell)$ in $T$ between $a$ and $b$ to give $u$ line $\ell$ in modified state, where $e_t(\ell)$ would set $t$'s permission based on whether $a$ happens before $b$. Thus if $a$ does not happen before $b$, then $e_t(\ell)$ sets $t$'s permission on $x$ to None and $b$ violates this permission, which is a contradiction.

If $d$ is a coherence event $e_t(\ell)$, then it must downgrade $t$'s permission on $x$ from Write to Read, as set at some prior memory operation $f$ by $t$, where $W_x^f \sqsubseteq C_t^f$ and $R_x^f \sqsubseteq C_t^f$. The downgrade must be due to a remote read operation, thus $R_x^b$ has changed from $R_x^f$ and possibly from $R_x^d$ if more remote reads $g$ follow the one that generated the coherence event, however the last-write vector clock has not changed as a result of the read. Any difference between $W_x^d$ and $W_x^b$ may only be due to writes by $t$, otherwise some coherence event would have downgraded $t$'s permission on $x$ to None. Writes by $t$ are ordered by program order with $b$, so in either case, it will be true that $W_x^b \sqsubseteq C_t^b$. For $t$'s Read permission on $x$ to allow $b$, $b$ must be a read operation, so the data-race check would pass if we ran it now (at event $b$), so $a$ must happen before $b$, which is a contradiction. Thus, in all cases, if $d$ sets permissions that allow $b$ then $a$ must happen before $b$. Local permissions are sound even when downgraded only on coherence actions.

#### 3.3.6.2 Proof: Local Updates Suffice for Permitted Accesses

Updates to access history on accesses to $x$ by thread $t$ that are allowed by local permissions must not require updates outside $\mathbb{W}_x(t)$ and $\mathbb{R}_x(t)$. This includes in-hardware status downgrades: accesses allowed by local permissions must not require in-hardware status downgrades for $x$. In-hardware status *upgrades* are never required for soundness; it is always safe to do a combined hardware-software check instead of a hardware check.

**Reads** of $x$ by thread $t$ update $\mathbb{R}_x(t)$ only. Reads to $x$ allowed by local permissions never cause eviction of metadata for $x$, so they do not downgrade in-hardware status. Although in-hardware status upgrades are never required, they would never be possible on reads allowed by local permissions. Upgrades on reads occur only after consulting the software handler, which is not invoked if local permissions allow the access.

**Writes** to $x$ by thread $t$ update $\mathbb{W}_x(t)$. While the canonical vector-clock algorithm may zero $R_x(u)$ $\forall u$ and $W_x(t)$ $\forall u \neq t$ as an optimization, this is not necessary to preserve soundness. Thus writes to $x$ allowed by local permissions can proceed with updates to at most $\mathbb{R}_x(t)$ $\mathbb{W}_x(t)$.

Nonetheless, on a system that zeroes other last reads and last writes on a new write, when a write is allowed by local permissions, it is guaranteed that $\forall u \neq t$, $R_x(u) = 0 \wedge W_x(t) = 0$. For the write to be allowed by local permissions, it must be the case that local WRITE permission was set on some previous write by thread $t$ that was not allowed by local permissions, at which time $R_x(u)\ \forall u$ and $W_x(t)\ \forall u \neq t$ were set to zero. The fact that the current write is allowed by local permissions guarantees that no other thread has accessed $x$ since this previous permission-setting, metadata-zeroing write and it remains true that $\forall u \neq, R_x(u) = 0 \wedge W_x(t) = 0$. Thread $t$ may have performed an intervening read, updating $\mathbb{R}_x(t)$, but an update to $\mathbb{R}_x(t)$ is allowed.

Writes set in-hardware status to EVERYTHING, which may be an upgrade, but never a downgrade. As discussed above, in-hardware status upgrades may always be omitted soundly. Nonetheless, in-hardware status is already EVERYTHING if all writes not allowed by local permissions set in-hardware status to EVERYTHING. By the same reasoning that showed zeroing persists from the previous write not allowed by local permissions, this previous permission-setting write must also have set EVERYTHING status which persists through this access. In-hardware status is only downgraded by evictions of metadata. Given that this access was allowed by local permissions, no other threads have accessed $x$ since the last permission-setting write to $x$ by $t$. Thus the only non-zero metadata for $x$ belongs to $t$. If $t$'s metadata for $x$ was evicted to cause an in-hardware status downgrade, then local permission would also have been cleared and this access would not be allowed by local permissions, which is a contradiction.

## 3.4 Related Work

We now discuss other related work on hardware support for dynamic data-race detection. Conflict Exceptions [73] and DRFx [78] are especially related to RADISH. They generate exceptions only for data races that may violate sequential consistency in data-race-free models. DRFx uses a hardware buffer of memory locations accessed between fences and coherence event monitoring that checks for conflicts with addresses in the buffer. Conflict Exceptions keeps pre-assigned byte-level access bits per cache line and sets aside memory space to keep access bits for out-of-cache data. Both proposals have large dedicated hardware structures: Conflict Exceptions adds 50% cache overhead, and DRFx adds 10KB of hardware state. The HardBound system [36] also leverages the idea of storing metadata in the cache data array, to provide memory safety for C programs. Aikido [95] uses dynamic binary rewriting and hardware memory protection to efficiently detect shared data, accelerating dynamic analyses such as data-race detection.

Min and Choi [82] developed a limited form of happens-before data-race detection using coherence events for programs with structured parallelism. SigRace [83] uses signatures to accelerate data-race checks; it employs a checkpoint/rollback mechanism to re-execute when a conflict is detected to prune some false positives. Still, SigRace can report false data races due to signature imprecision and granularity of access monitoring, as well as missed data races due to limited buffer space for checkpoint/rollback. CORD [103] approximates happens-before data-race detection using per-word vector clocks (fixed metadata) for in-cache data only, leading to both unsoundness and potential incompleteness. ReEnact [104] uses thread-level speculation mechanisms to detect data races and potentially recover from them via checkpoint/rollback. ReEnact can report false data races due to word-granularity tracking and can miss data races due to finite hardware resources.

HARD [143] is a hardware-based implementation of the lockset algorithm that uses Bloom filters per cache line to encode which locks should be held when accessing the corresponding data. While locking-discipline violation detection is very useful for debugging, it is imprecise for many acceptable programming idioms.

ECMon [85] proposes exposing cache coherence events to software as a primitive for several program monitoring and control techniques. ECMon has a subset of the support that Radish offers, as we need hardware for byte-level metadata tracking and vector-clock comparisons, since trapping to software for every metadata update and every vector-clock computation would be prohibitively expensive.

## 3.5 Conclusions

Our work on Radish showed that accurate hardware-supported data-race detection is feasible by using hardware support to optimize common cases of a full, accurate software data-race detector, while retaining software support to handle the rare cases that are a poor fit for fixed hardware resources but necessary for full accuracy. We showed the accuracy Radish by its equivalence to the canonical vector-clock algorithm. The prospect of an accurate, fast, and general hardware-supported data-race detector is promising, but Radish's accuracy remains limited to ISA-level programs. We address this problem in the next chapter.

Chapter 4

# LARD: Low-Level Abstractable Race Detection

*Low-level Abstractable Race Detection (LARD)* virtualizes low-level dynamic data-race detection to detect language-level data races. This chapter presents work originally published in the proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems [137].

## 4.1 Introduction

Recent proposals for low-level dynamic data-race detection have full accuracy and improved performance. RADISH (Chapter 3 and [38]) uses a mix of software and hardware support for data-race detection fast enough for many deployment situations. Aikido [95] uses hypervisor support and page-protection to accelerate software analyses. These are *low-level data-race detectors*: they analyze virtual memory accesses in the instruction set architecture (ISA) and store access history for virtual memory locations.

Low-level—and specifically hardware—support can **improve performance** and accelerate checking in common cases where no data-sharing occurs. These systems exploit the fact that memory accesses can be tracked efficiently with hardware or hypervisor support and are much more frequent than synchronization operations. They can also implement the critical core of data-race detection logic once in a **general**, reusable, low-level mechanism. Although a low-level mechanism *alone* is insufficient for language-level data-race detection, it can *reduce* the complexity of high-performance data-race detection in a language implementation. Our implementation of LARD (§4.3) shows that the engineering required is feasible for a system as complex as a JVM. For unmanaged targets like C programs, it is simpler.

Naïvely, one might run an unmodified low-level dynamic data-race detector "underneath" a high-level language implementation like a Java virtual machine (JVM) to detect *language-level data races* in the Java program. By *language-level data race*, we mean a data race between accesses in the high-level language memory abstraction. This work focuses on detecting these (and only these) data races.

We present the first full treatment of why an unmodified low-level data-race detector *does not work* to detect language-level data races. We then develop extensions to low-level data-race detection and high-level language implementations to remove all sources of missed data races and false data races. Finally, we implement and evaluate a prototype hardware-based dynamic data-race detector for Java, showing that our extensions enable accurate data-race detection for high-level languages using general low-level hardware support.

### 4.1.1 Low-Level Data Races $\neq$ Language-Level Data Races

*Neither low-level data races nor language-level data races subsumes the other.* Some low-level (*e.g.* x86) data races are not language-level (*e.g.*, Java) data races and some language-level data races are not low-level data races. Thus *a low-level data-race detector reports false data races and misses true data races for programs written in high-level languages.*

Consider a Java program running on a JVM on a multicore processor with data-race detection that analyzes all memory loads and stores for data races. The Java abstraction of execution, with field accesses and lock operations, is fundamentally different from the low-level abstraction of instructions accessing virtual memory, as analyzed by the data-race detector. Two broad features of the translation cause false and missed language-level data races for low-level data-race detectors.

First, low-level executions contain instructions not derived from equivalent—or *any*—operations of the language-level execution. Memory accesses and synchronization in implicit JVM services like garbage collection are not derived from explicit operations of a Java program. Some low-level memory accesses implement language-level synchronization operations.

Second, language-level semantics of low-level resources change during execution. The allocator or garbage collector reuses memory or moves objects in memory. By analyzing accesses to a single virtual memory location reused by the JVM to store distinct Java objects, a data-race detector can report false data races. By analyzing distinct virtual memory locations when a single Java object has moved, the detector can miss true data races. If a threading implementation multiplexes language-level threads on low-level threads, two operations of the same low-level thread do not necessarily belong to the same language-level thread, and vice versa, leading to false or missed data races subject to thread placement.

### 4.1.2 Low-Level Detection of Language-Level Data Races

This work's main contribution is *low-level abstractable race detection* (LARD), an extended low-level data-race detector interface that lets run-time systems and compilers communicate a language-level view of execution in sufficient detail. The extensions are minimal: they let run-time systems and compilers (1) mark language-level memory accesses and synchronization operations explicitly for analysis, leaving system operations unanalyzed, (2) report changes in the language-level/low-level memory mapping due to memory reuse and movement, and (3) report language-level thread identity. The low-level data-race detector then analyzes only the memory accesses and synchronization operations of the language-level program and updates its state according to changes in memory allocation to reflect the language-level memory abstraction.

LARD allows data-race detectors for high-level languages to harness the performance and generality of low-level detection mechanisms while maintaining accuracy. The focus of this work is a simple execution stack consisting of a language run-time running on hardware, but the design generalizes to a range of dual-level execution environments such as operating systems and hypervisors.

To the best of our knowledge, this work presents the first design for *virtualizing data-race detection*. Earlier exclusively low-level or language-level data-race detectors (*e.g.*, RADISH (Chapter 3), Helgrind [125], and FastTrack [52]) have support for marking custom synchronization routines to avoid false data races on these accesses and track their synchronization effects. Low-level detection of language-level data races requires similar support, but may also need to distinguish the semantics of such operations based on context.

Some C data-race detectors (*e.g.*, [118, 125]) treat allocation specially to help reduce false data races and some JVM data-race detectors work around object movement to avoid missed data races (*e.g.*, [30, 105]), but these previous efforts have focused mainly on ad hoc *reduction* of imprecision in fundamentally imprecise data-race detection algorithms implemented at a single level of abstraction. In this work, we characterize the effects of the language-level/low-level translation on data-race detection in depth and develop a unified approach to *eliminate all* missed and *all* false data races, achieving accurate language-level data-race detection in high-level programs using low-level data-race detectors.

### 4.1.3 LARD Implementation and Evaluation

To evaluate the feasibility of low-level abstractable race detection, we implemented an accurate data-race detector for Java using LARD. Our implementation includes two independent systems, communicating only via LARDx86, an extension of the x86 ISA with flags to mark memory access instructions explicitly to be checked for data races and instructions to report synchronization, memory reuse, movement, and thread identity. We simulated an accurate, low-level, mostly-hardware, dynamic data-race detector based on RADISH (Chapter 3 and [38]), and extended to support LARDx86. We modified Jikes RVM [8] to emit data-race-checked accesses for application code and unchecked accesses for JVM code, report synchronization events in application code, and report memory reuse and movement events in the JVM. Our results show that our extensions are necessary and sufficient to avoid false and missed data races in practice.

### 4.1.4 Contributions and Outline

- We explain why low-level data-race detectors are incorrect for high-level programs (§4.2), synthesizing disparate issues encountered in prior contexts [30, 105, 118, 136].

- We design *low-level abstractable race detection* (LARD), a simple interface for low-level data-race detectors and language implementations that enables accurate language-level data-race detection using a low-level data-race detector, and compare our approach to some prior systems (§4.2).

- We implement our approach for Java, coupling a simulated hardware-supported ISA-level dynamic data-race detector and a modified Jikes RVM through a version of the x86 ISA extended with LARD primitives (§4.3).

- We evaluate our implementation's accuracy, comparing against FastTrack [52], a naïve low-level data-race detector similar to RADISH (Chapter 3), and various partial implementations of LARD, finding that, *in practice*, naïve ISA-level data-race detectors suffer from false and missed data races for Java programs, but LARD does not (§4.4).

Finally, we discuss more related work (§4.5) and conclude (§4.6).

## 4.2 Low-Level Abstractable Race Detection

*Low-level abstractable race detection* (LARD) extends the interface and functionality of low-level data-race detectors to abstract (*i.e.*, virtualize) data-race detection to high-level execution environments. The key idea is to preserve relevant information from language-level operations in the low-level execution. LARD requires cooperation from the high-level language implementation and the low-level data-race detector to implement two types of extensions: (1) distinguish between source operations and run-time system operations; and (2) maintain the mutable mappings from language-level memory to low-level memory and from language-level threads to low-level threads.

We explain LARD by examining the five fundamental operations where relevant differences arise between language-level and low-level views of execution: memory access (§4.2.1), synchronization (§4.2.2), memory allocation (§4.2.3), memory movement (§4.2.4), and thread mapping (§4.2.5). For each operation, we show how information lost in translation can cause missed or false data races. Then we extend the low-level data-race detector interface to retain sufficient language-level information. Table 4.1 summarizes the five interface extensions. Finally, we argue that LARD's five extensions are sufficient to virtualize general low-level

| **Operation** | **Prevents** | **Translation Issue** |
|---|---|---|
| (un)tracked access | false data races | program vs. system |
| (un)tracked sync. | missed data races | program vs. system |
| clear history | false data races | memory reuse |
| move history | missed data races | memory movement |
| set thread identity | false and missed data races | thread scheduling |

**Table 4.1:** The LARD interface.

data-race detection for use by language implementations (§4.2.6) and discuss how LARD's design principles generalize to other environments (§4.2.7).

We focus on Java programs executed by a modern JVM on hardware that detects ISA-level data races, although we believe that these five issues arise in any setting where data-race detection is implemented at a significantly different abstraction level than the source program. Others have previously identified or mitigated some of these issues. (We discuss related work inline and in §4.5.) However, we believe our research is the first to consider the full set of techniques needed to virtualize data-race detection.

### 4.2.1 Memory Access

JVM execution contains both memory accesses compiled from explicit field and array accesses in Java programs and accesses in the JVM itself. Observing the latter may cause the detector to report data races involving at least one JVM access. These are false Java data races because they involve access outside the Java program and its execution abstraction.

**Rule 1** *The data-race detector should check only accesses explicit in the source program for data races.*

**Examples** The implementation of locks, for example, is necessarily lock-free, using memory reads and writes plus hardware synchronization primitives like fences and atomic compare-and-swap. These memory operations may race, but are chosen carefully with respect to the hardware memory model to ensure correct behavior regardless. The need to ignore these *synchronization races* is well-understood.

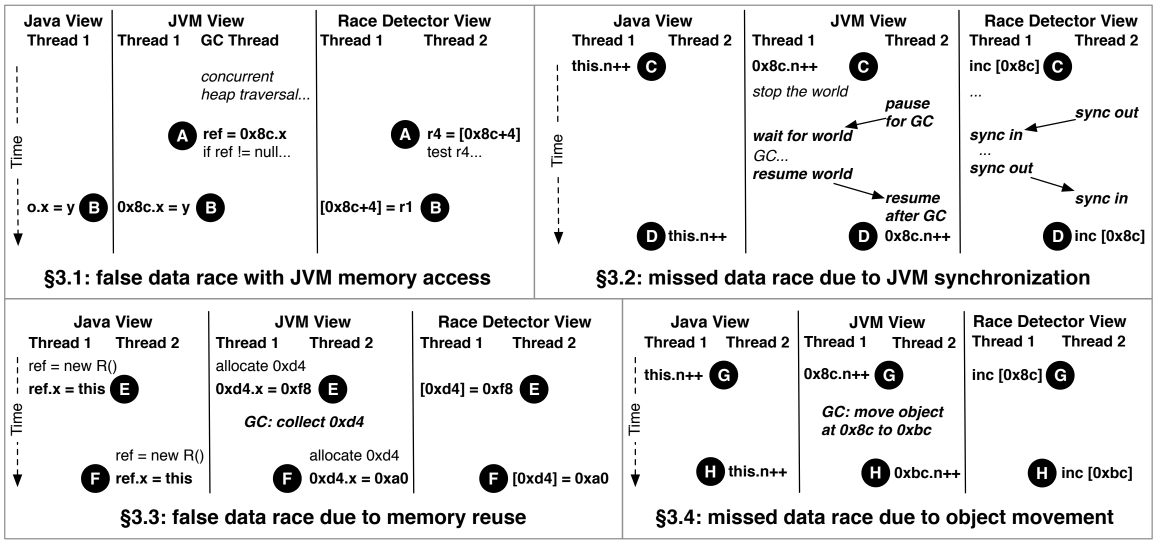


**Figure 4.1:** A naïve low-level data-race detector reports false data races and misses true data races in Java programs. Solid arrows are happens-before edges observed by the data-race detector. Each example contains three views of the same execution.

Consider JVM accesses to the Java heap. A concurrent mark-sweep garbage collector, for example, traverses the heap while Java threads continue to mutate it, compensating in a safe, algorithm-specific way, for the data races that are bound to result. Figure 4.1 shows an example: A GC thread reads o.x (stored in memory at address 0x8c+4) at A during a concurrent heap traversal. A low-level data-race detector will report a data race with this read on Thread 1's later write at B.

Since data-race detector access histories typically store only the *last* write by any thread and *last* read by each thread, a false data race can overwrite history necessary in the future to detect a true data race with a previous program access.

**Extension** It is natural to distinguish source-program and run-time system accesses with explicit *tracked* and *untracked* access instructions. Both have the conventional functionality of memory accesses. The low-level data-race detector analyzes tracked accesses only.

### 4.2.2 Synchronization

Witnessing implicit JVM synchronization where no Java synchronization exists can cause the data-race detector to miss true data races in the Java program: the racing accesses appear well-ordered to the low-level data-race detector. A data-race detector that observes *all* synchronization never misses a data race that may cause a sequential consistency violation, because it only misses data races due to *extra* synchronization, but it can miss other problematic data races.

**Rule 2** *The data-race detector should analyze only synchronization explicit in the source program.*

**Example** Consider the upper left example in Figure 4.1, which shows views of the same multithreaded execution at the Java, JVM implementation, and low-level data-race detector levels of abstraction. In the Java view, Threads 1 and 2 both increment the `n` field of the same object at `C` and `D`, respectively. Neither thread synchronizes, so the accesses race. A low-level data-race detector misses this data race if a stop-the-world garbage collection occurs between the two accesses. If the data-race detector witnesses the global barrier synchronization, accesses `C` and `D` appear well-ordered. This Java data race is not a low-level data race in this execution, due to this JVM synchronization, but it could be in another execution depending on the timing of garbage collection.

**Extension** Accurate ISA-level dynamic data-race detectors such as RADISH (Chapter 3) provide primitives for synchronization libraries to report happens-before effects of synchronization to the data-race detector. This approach is easily adaptable for LARD: synchronization reports should be issued only for synchronization operations explicit in the source program.

### 4.2.3 Memory Allocation

Without knowledge that a low-level memory location has been reused to store a new, distinct language-level memory object, a low-level data-race detector may report false data races between accesses to the new and old language-level occupants.

**Rule 3** *The data-race detector should clear access histories of low-level memory locations atomically with collection or freeing of their language-level occupants.*

**Example** Consider the example in the lower left of Figure 4.1. In the Java view, Thread 1 allocates an `R` object then writes to its `x` field at `E`. Later, Thread 2 allocates a distinct `R` object then writes to its `x` field at `F`. Clearly these Java accesses to fields of distinct objects cannot race, but a low-level data-race detector may report a data race between the writes at `E` and `F` if the memory manager reuses the same memory to store these two objects and threads accessing these objects do not synchronize, as shown in Figure 4.1. The data-race detector ignores memory accesses and synchronization in the garbage collector, per Rules 1 and 2, and reports a data race on the concurrent, conflicting writes to address `0xd4`.

**Extension** Synchronization in the memory manager that makes reuse of memory across threads must not be observed by the data-race detector, since it could also hide true data races in the Java program (§4.2.2). In the C/C++ memory model [26], freeing a location *happens before* a later allocation of the same memory location, but the ordering applies only to accesses to that memory location.

We choose to clear low-level data-race detector access histories on deallocation. Thus newly allocated low-level memory locations appear fresh to the data-race detector whether or not they have been accessed before. This approach matches the language-level memory abstraction and is safe given memory safety, making it valid for managed environments and well-behaved C/C++ programs. It is the responsibility of the data-race detector to clear access histories when requested. The run-time system must use this support to ensure it clears access histories of a memory location before reallocating it. Helgrind [125] and RACEZ [118], two data-race detectors for C programs, take a similar approach, treating malloc/free specially. Helgrind additionally provides C preprocessor macros to mark custom allocation routines to help reduce false data-race reports.

### 4.2.4 Memory Movement

Some garbage collectors move language-level memory objects in low-level memory during collection to defragment the heap. Movement is not part of the language-level memory abstraction. Collectors update all references to moved objects to maintain a consistent heap. Without knowing a language-level memory object has moved, a low-level data-race detector may miss true data races between accesses at the object's old and new low-level memory locations. Choi, *et al.*, briefly identified (but did not solve) one aspect of this problem in [30], §3.3. Similar issues arise for an operating system or hypervisor running above a physical-memory data-race detector when remapping virtual-memory pages.

**Rule 4** *When the run-time system moves a language-level object from one low-level memory location to another, the data-race detector should move the corresponding access histories along with the objects.*

**Example** Consider the example execution in the lower right of Figure 4.1, in which two Java threads increment the `n` field of the same Java object, with no synchronization. Clearly, this is a Java data race. However, garbage collection is triggered after Thread 1 increments `n` at `G` but before Thread 2 does at `H` and the garbage collector moves shared object at address `0x8c` to address `0xbc`. The low-level data-race detector ignores accesses and synchronization by the collector, as it should. Because the object moves, the two threads access different low-level memory locations and the data-race detector misses a true Java data race.

**Extension** Moving a language-level object in low-level memory may be a non-atomic operation, requiring copying the contents of several low-level memory locations. Some concurrent copying collectors (*e.g.*, [100]) may begin movement operations optimistically, abort midway, or copy a single low-level memory location multiple times to support continued non-blocking access by program threads during object movement. Since object movement is rarely implemented by a single, atomic, low-level operation and clearing access history is already required to support memory allocation, it is natural to add a primitive to *copy* the access history for one low-level memory location to another. Once the language-level object is fully copied, the old copy can be deallocated, at which point its access history is cleared. Garbage collectors already ensure that the movement of data appears atomic; it is also their responsibility to ensure that they request access history movement from the low-level data-race detector in a way that ensures access history is moved atomically with program data. Neither the run-time system nor the data-race detector can accomplish this alone.

An alternative is to use logical addresses for data-race detection analysis so it is resilient to movement [105], but this adds a logical address lookup indirection to the critical path of memory accesses, along with the cost of managing available unique identifiers. For low-level data-race detectors, each tracked memory instruction in the ISA would need to take an extra logical address argument. TLB-like hardware support to cache the low-level to logical address mapping could speed the lookup at the cost of lengthening the critical path for cache accesses, which is unmodified in hardware data-race detectors such as Radish (Chapter 3), and would require shootdown on object movement anyway. While reporting movement has overheads to copy access histories, data-race-checked memory accesses occur *far* more frequently than object movement. We choose reporting movement (copying) as a less invasive and higher-performance option.

### 4.2.5 Thread Identity

Some threading implementations (*e.g.*, user-level threads or work-stealing schedulers) multiplex a set of language-level threads on a fixed set of low-level (kernel or hardware) threads. Without knowledge of this mapping, a low-level data-race detector may report false data races between accesses in a single language-level thread or miss true data races between accesses in distinct language-level threads.

**Rule 5** *The data-race detector should use language-level thread identities in its analysis.*

**Example** When two language threads execute conflicting accesses without synchronization, they clearly race. However, if they execute their accesses while scheduled on the same kernel thread (at different times), the low-level data-race detector observes two accesses by the same kernel thread and misses a true data race. When a single language thread executes multiple accesses to the same location, it cannot race with itself, but if these accesses are executed while it is scheduled on distinct kernel threads (at different times), the low-level data-race detector observes conflicting accesses between distinct kernel threads and may report a false data race.[1]

**Extension** We take an approach similar to object movement, reporting a new thread identity whenever a threading system schedules a language-level thread onto a low-level thread. The low-level data-race detector uses this thread identity for all operations of the low-level thread until the next such report.

### 4.2.6 Sufficiency

Data races—and accurate algorithms for their detection—are defined in terms of memory, synchronization, and threads. Program translation must affect one of these features to affect post-translation data-race detection. To derive the set of translation issues affecting data-race detection, we enumerated all differences introduced in the language-to-ISA translation that interact with these features, to the best of our knowledge, finding that only the issues in §4.2.1-4.2.5 affect data-race detection. The five extensions described above ensure data-race detection is performed only on language-level memory and synchronization operations (§4.2.1, §4.2.2) and on the language's abstractions of memory (§4.2.3, §4.2.4) and thread identity (§4.2.5). While each issue and extension is fairly simple, it is their composition that allows data-race detection to virtualize. We believe our research is the first to consider the full set of techniques needed to virtualize data-race detection. This perspective was essential for guiding our implementation and evaluation.

LARD's effectiveness depends on the contract between the low-level data-race detector and the language implementation; it does *not* free the language implementation from reasoning about all details of data-race detection. It is up to the language implementation to ensure that its use of the LARD primitives meets its particular semantics. For example, garbage

[1]We assume the data-race detector ignores synchronization involved in the context-switching of language threads, as it should.

collectors are responsible for ensuring that primitives like access history clearing and copying appear atomic with respect to memory accesses by program threads.

The language implementation is also responsible for compilation choices. For example, some compiler optimizations allowed by the Java [76] and C/C++ [20] memory models may remove data races from the original program, but none introduce data races where they did not exist. *Roach-motel reordering* allows the movement of memory operations into—but not out of—critical sections [133]. As a result, the access may now be well-ordered when it would have raced if the transformation was not applied. We do not consider this a missed data race. It is explicitly allowable behavior in the language memory model and, unlike with the issues above, the program can never manifest this data race as compiled. If the language implementers *do* consider this a missed data race, they must choose compiler optimizations appropriately. Regardless, this is the language implementer's choice and is an issue common to dynamic data-race detectors implemented at *all* levels of abstraction, not just low-level data-race detection for high-level languages.

### 4.2.7 Generality

This work has focused on a simple execution stack consisting of a language implementation runinning on hardware with accurate data-race detection support. We believe the design principles of Lard also apply to other environments involving translation between two abstractions of shared-memory multithreading. For example, consider the implementation of a C-level data-race detector using hypervisor support or a hardware data-race detector that reasons in terms of physical (not virtual) memory addresses. The intervening virtual memory layer provides an abstraction of memory by translating down to the hypervisor or hardware below in ways manifesting the five issues discussed above. Virtual memory paging, for example, exemplifies the same issues raised by automated memory management. A task-based programming model implemented with work-stealing [16] on top of a shared-memory multithreaded language needs to handle the scheduling of tasks on threads and the synchronization introduced by a work-stealing system.

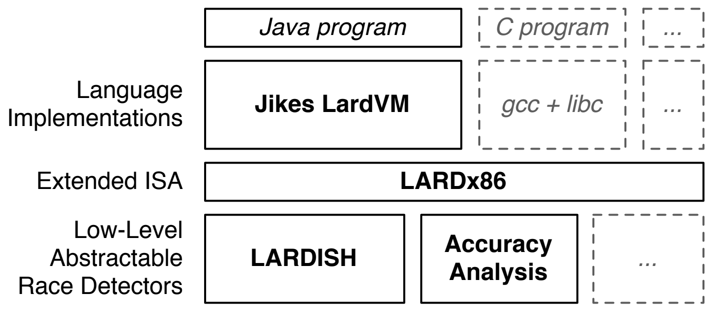


**Figure 4.2:** The LARD environment.

## 4.3 Implementation

To validate the efficacy and feasibility of low-level detection of language-level data races, we implemented a data-race detector for Java using a low-level data-race detector and a Java virtual machine that communicate through the LARD interface. This section describes four parts of our implementation (bold items in Figure 4.2): LARDx86 (§4.3.1) is an extension of the x86 ISA with LARD primitives. LARDISH (§4.3.2) is a simulated hardware implementation of LARDx86 that performs accurate, LARD-aware, data-race detection derived from the RADISH (Chapter 3) hardware-based data-race detector. Jikes LARDVM (§4.3.3) is a Java virtual machine that runs on LARDx86 and implements accurate Java data-race detection using the LARDx86 primitives. We also extended these three parts for fine-grained accuracy evaluation of LARD and naïve low-level data-race detectors (§4.3.4).

### 4.3.1 The LARDx86 ISA

LARDx86 extends the x86 ISA to provide a LARD interface between software run-time systems and low-level vector-clock data-race detectors. To support the tasks described in §4.2, LARDx86 extends the x86 ISA with Tracked accesses, a Thread instruction to report thread identity, WriteVC and ReadVC instructions to report synchronization, and ClearHistory and CopyHistory to manipulate access histories.

**ISA Extensions for Memory Access and Synchronization** Explicit Tracked memory access instructions, distinguished by separate opcodes or a prefix, have the usual semantics of memory accesses, but are additionally checked for data races by the low-level data-race detector. Untracked accesses are never checked. The low-level data-race detector must store a vector clock for each thread internally, representing the most recent event from each thread that happens before the current event of this thread. It is used when checking Tracked accesses for data races. The run-time system must track ordering effects of synchronization using vector clocks, using the ReadVC and WriteVC instructions to read and write entries in the low-level data-race detector's per-thread vector clocks.

Alternatively, the run-time system could report synchronization on a particular memory object (*e.g.*, lock), leaving storage and tracking of all vector clocks to the low-level data-race detector. This hides vector clocks from the run-time system, but is best suited for locks or barriers. Additional purpose-built instructions would be necessary to support synchronization operations that also atomically manipulate data, such as language-level atomic CAS operations or Java's volatile field accesses (§4.3.3). Since the semantics of synchronization is best understood in its implementation (*i.e.*, above the ISA), exposing vector clocks is more flexible and ultimately has lower complexity.

**ISA Extensions for Mapping Memory Management** The ClearHistory and CopyHistory instructions clear and copy low-level data-race detector access histories of given memory locations. They are intended for use by a memory manager when it frees or moves language-level memory objects. ClearHistory takes two arguments, the address and size of a memory region whose access histories should be discarded. CopyHistory takes three arguments, the address of a source region of memory, the address of a destination region of memory, and the size of the two regions. (They must be the same size.) In response to a CopyHistory instruction, a low-level data-race detector copies the access history for each memory location in the source region to use as the access history for the corresponding memory location in the destination region. Specifying regions of memory rather than individual locations allows the data-race detector to optimize bulk updates internally, but regions are restricted to a single virtual memory page to simplify support in hardware-based data-race detectors.

**Encapsulated vs. Exposed Access Histories** LARDx86 assumes that the low-level data-race detector is solely responsible for managing access histories. Exposing control of access history storage to a language implementation may be more natural and efficient if access histories can be colocated with the data they shadow and automatically moved and deallocated by the garbage collector. This is not feasible with our implementation because the hardware data-race detector manages access histories based on physical—not virtual—addresses (§4.3.2). Reverse address translation or virtually addressed caches would be necessary to support efficient external software management of access histories.

### 4.3.2 The LARDISH Hardware Data-Race Detector

To evaluate the performance potential of a hardware-based LARD system, we designed LARDISH, an extension of RADISH (Chapter 3), a hardware-supported data-race detector. This section outlines our extensions to RADISH to support low-level abstractable data-race detection with LARDx86.

**LARD Extensions** Supporting LARD requires minor adjustments to RADISH's access and synchronization tracking and modest hardware extensions to support LARD's memory management. RADISH checks for data races on all memory accesses except stack accesses and accesses in its software manager. We generalize this to use LARDx86's explicit Tracked memory access instructions. We implement the WriteVC and ReadVC instructions with RADISH's existing support for tracking and reporting synchronization in libraries.[2] The Thread instruction is identical to a context switch in RADISH, swapping the thread's vector clock and requiring eager or lazy flushing of the old thread's cached access histories to the software manager, which maintains the mapping between processor cores, kernel threads, and language threads. ClearHistory and CopyHistory instructions must perform tasks similar to context switches or virtual memory paging in RADISH. Manipulating access histories in RADISH's software manager is straightforward when they are not cached in hardware, but hardware support is needed to invalidate or move cached access histories.

[2]The original RADISH simulator hard-codes pthreads support; we implement true vector-clock instructions.

**Manipulating Cached Access Histories** The RADISH software manager identifies access histories by physical addresses, since it receives physically addressed access history cache lines on eviction. We use the existing address translation mechanism to map the virtual addresses specified by ClearHistory and CopyHistory instructions to physical addresses that the extended RADISH software manager can use to manipulate the relevant access histories.

When a hardware cached access history is not available to perform a check, the RADISH hardware requests it from software. When dirty cached access history is evicted, RADISH calls its software manager to persist it instead of storing it to memory. The software manager keeps track of what access histories may be in-cache (on calls for fills and evictions). If an access history to be cleared or copied is not cached, it can be handled by the software manager alone. If an access history is cached, ClearHistory and CopyHistory must explicitly invalidate or copy hardware-cached access histories to ensure they stay consistent with the software-managed copies and the new mapping of language-level to low-level memory. For ClearHistory, we instead invalidate the cached access history without calling the software handler.

CopyHistory must persist hardware-cached histories for its source location. In this case, hardware first forces eviction of the source locations' cached access histories via the normal RADISH eviction handler. We assume that the destination of an object movement has no preexisting access histories. If it does, the memory manager must use ClearHistory to explicitly invalidate any cached version before copying. Once the software manager has persisted any cached access histories for the source region, it copies all access histories for the source region to become access histories for the destination region. On the first access to a location in the destination region after copying, hardware will request the software-managed access history as usual.

If the language implementation allows program threads to access objects concurrently with copying, the software manager must ensure the atomicity of the copy operation by disallowing hardware requests for access histories of the affected regions for the duration of the copy, effectively blocking access to the region during access history copying. To allow finer-grained access, concurrent copying collectors may issue smaller CopyHistory operations, ensuring atomicity of the full copy themselves.

An alternative is to copy cached access histories directly in cache, moving stale software-managed access histories to follow. This approach must consider whether it is profitable to keep the history in cache, but evict another cache line at its destination. We implement a simpler evict-and-copy policy. Additionally, a MoveHistory instruction could enable optimizations when the intent is to copy an access history and immediately clear the original, as in stop-the-world copying collectors. We have not explored this.

### 4.3.3 The Jikes LARDVM Java Virtual Machine

Jikes LARDVM is a modified version of Jikes RVM [8] 3.1.1 that implements accurate data-race detection for Java using LARDx86. Jikes LARDVM marks source program accesses for data-race checks (§4.3.3.1), tracks source program thread identity and synchronization (§4.3.3.2), and reports garbage collector operations (§4.3.3.3). Our implementation was designed for detailed accuracy analysis and is only lightly optimized for performance.

#### 4.3.3.1 Memory Tracking

The Jikes LARDVM JIT compiler emits Tracked accesses for potentially racing field and array accesses in Java programs. All other accesses in the system are unmarked, to be ignored by the low-level data-race detector. This includes JVM code like garbage collector write barriers that are inlined into compiled code. In write barriers, the memory write on behalf of the source program is marked Tracked, but all other accesses are untracked. The compiler may decide not to mark some source program accesses Tracked if it can prove their data-race freedom, such as for read-only final fields. We have not yet enabled thread-escape analysis.

Since Jikes RVM is a self-hosted JVM written in Java and compiled to machine code by its own compiler, the compiler must distinguish Jikes RVM Java code from source program Java code, emitting Tracked accesses and synchronization reporting only for the source program. Compilers in conventional JVMs written in lower-level languages can emit Tracked accesses everywhere except inlined VM code.

#### 4.3.3.2 Thread Identity and Synchronization

Jikes RVM uses kernel threads directly, so we issue a Thread instruction only once per thread. Every Java object may be used as a lock. To track lock synchronization, we shadow each lock with a vector clock indicating the last logical time it was released. A word in the object header stores a pointer to a lazily allocated vector clock. We also augment each Jikes RVM thread with a thread vector clock tracking its logical time and synchronization with other threads. We instrument lock operations and thread fork/join to track happens-before ordering with vector clocks in the JVM, reporting updates to the data-race detector's per-thread vector-clock using WriteVC/ReadVC. Jikes RVM and source programs share synchronization implementations. We added an instrumentation-wrapped version of each for program synchronization.

Under the Java Memory Model [76], volatile field accesses never race; they induce synchronization instead. There is a happens-before edge from a volatile write to each volatile read observing the write, enforced by hardware memory fences and restrictions on compiler reorderings. Jikes LARDVM never marks volatile accesses Tracked. Each volatile field is shadowed by a vector clock representing the last logical time a volatile write occurred on that field. The field contents and the vector clock must be updated and observed atomically. Volatile reads have the same type of happens-before effects as lock acquires (volatile writes are like lock releases), but provide no mutual exclusion. The vector clock to store on a volatile write operation depends on the field's vector clock at the time of the write, which requires allocating one new vector clock per volatile write operation. To avoid introducing a lock into the source program's lock-free code, we align volatile fields with an adjacent vector clock pointer and use a wide CAS to operate atomically over the two, optimistically computing the vector clock to store and retrying under contention in a standard lock-free manner.

#### 4.3.3.3 Memory Management and Mapping

We modified the classical mark-sweep and semispace collectors in Jikes RVM and MMTk [13] to issue a CopyHistory instruction when moving an object and a ClearHistory instruction when reclaiming an object, including after movement. The garbage collector handles source program objects and objects representing Jikes RVM internals (including vector clocks). We issue ClearHistory and CopyHistory conservatively for all objects whose contents *may* have been used by data-race-checked source program accesses and thus may have access histories in the data-race detector. Unlike synchronization and access, conservative reporting of memory-management events cannot cause missed or false data races.

Other more advanced garbage collectors do not introduce other issues beyond the reuse and movement exhibited by mark-sweep or semispace. For example, accesses to metadata in generational write barriers are properly left untracked, as they are not explicit in the program.

#### 4.3.3.4 Extent of Changes to Jikes RVM

Our modifications to Jikes RVM add or change 8229 lines of code. The core LARD additions account for well under half those lines, with the rest devoted to extensions for accuracy analysis (§4.3.4), extensive debugging/profiling, and significant trivial code duplication for garbage collector access barriers.

The core LARD additions mainly track synchronization events and mark tracked accesses in the compiler. Other significant additions involved object layout for volatile fields with adjacent vector clocks and proper garbage collector tracing of vector clock references. Those aspects of the data-race detector that are shared in common with software implementations (*i.e.*, synchronization tracking) were the most complicated to implement. Memory reuse and movement reporting was relatively simple once we understood the memory management architecture in Jikes RVM and MMTk.

### 4.3.4 Extensions for Accuracy Analysis

For the accuracy analysis in §4.4.2, we built an analysis tool that runs multiple data-race detection algorithms on the same LARDx86 execution, comparing their results at each memory access. Specifically, we support several detectors that each ignore one of the LARD extensions in their analysis. It is simple to analyze all accesses instead of Tracked accesses only. To ignore memory reuse or movement, a detector ignores the ClearHistory or CopyHistory instructions, respectively. To analyze all synchronization instead of language-level synchronization only, we must explicitly report all synchronization. Tracking all synchronization in Jikes LARDVM on the same execution where we track language-level synchronization requires a separate vector clock for each lock, volatile field, and thread. This addition is needed only for the accuracy evaluation (§4.4.2) and not in real deployments.

## 4.4 Evaluation

We evaluate the efficacy of LARD and multiple naïve low-level data-race detectors for implementing accurate dynamic data-race detection for Java. We validate LARD's ability to eliminate missed data races and false data races by comparing the results of our LARD-based Java data-race detector with the accurate FastTrack Java data-race detector [52] and a naïve low-level vector-clock data-race detector (§4.4.1) and quantify the effects of individual LARD extensions in terms of the missed data races and false data races they eliminate (§4.4.2). Although the accuracy analysis is the main contribution of our evaluation, we include initial evaluation of the performance of a LARD implementation composed of Jikes LARDVM executing on LARDISH via simulation (§4.4.3). Additionally, we evaluate the performance of Jikes LARDVM alone on conventional x86 hardware (§4.4.4).

**Experimental Configuration** We ran experiments with multithreaded benchmarks from Java Grande [122] and DaCapo 9.12 [14].[3] The Java Grande benchmarks are mostly small scientific applications with relatively static memory footprints.We replace Java Grande's custom racy spin-waiting barriers with data-race-free versions to verify LARD's accuracy for data-race-free programs. We report their results as one unit. DaCapo is composed of larger applications with varied concurrency patterns. Where applicable, benchmarks were configured to use 4 threads. To make heavyweight accuracy analyses and simulations feasible, we used small inputs. For performance experiments on real hardware, we used the largest inputs. We ran two configurations each of Jikes RVM and Jikes LARDVM, using the mark-sweep (MS) and semispace (SS) garbage collectors. All experiments were run on

[3]We omit DaCapo benchmarks that crash on unmodified Jikes RVM or take too long to complete under simulation.

quad-core 64-bit 2.8GHz Intel Xeon Pentium 4 machines with 4GB of RAM and a Linux 2.6.32 kernel.

### 4.4.1 False Data Races and Missed Data Races

To validate the accuracy of our LARD implementation and illustrate the degree to which naïve low-level data-race detectors can suffer from false and missed data races, we compare the data-race reports from three data-race detectors. **FastTrack** [52] is an accurate data-race detector for Java implemented via bytecode instrumentation. We compared FastTrack data-race reports with those of our **LARD** implementation (described in §4.3) and a **Naïve** low-level vector-clock data-race detector equivalent to RADISH (Chapter 3). LARD and Naïve are run over the same exact execution of the benchmarks, but FastTrack uses separate executions. We have not reimplemented FastTrack within Jikes LARDVM due to the complexity of co-hosting the two. FastTrack ignores accesses and synchronization in the JDK standard libraries. LARD instruments all Java code by default.

Table 4.2 contains results from running these data-race detectors on the Java Grande benchmarks and selected DaCapo benchmarks. Exact numbers can vary across executions; we show the highest number of data races reported by each detector on any *single* execution. Column **DRE** shows the number of dynamic accesses on which data races were reported. Column **PC** shows the number of distinct program counters of these reports. Each racy access (DRE) races with one *or more* previous accesses, hence counting DREs is slightly different than counting distinct *data races*. We report data races on the second of a racing pair of accesses and only report those racing pairs where the previous access has happened since (or is) the last write to an address. Due to this optimization, FastTrack and LARD are accurate (sound and complete) through the first data race. In practice, accuracy is not often compromised thereafter, so we report total data-race reports in each execution as is conventional in the literature.

**FastTrack vs. LARD** We compared data-race reports from FastTrack and LARD to validate LARD's accuracy. LARD consistently reported at least as many data races and at least as many racy accesses as FastTrack, though generally in the same order of magnitude.

| **Benchmark** | **FastTrack** | | **GC** | **LARD** | | **Naïve** | |
|---|---|---|---|---|---|---|---|
| | **DRE** | **PC** | | **DRE** | **PC** | **DRE** | **PC** |
| Java Grande | 0 | 0 | MS | 0 | 0 | 28211 | 124 |
| | | | SS | 0 | 0 | 7899 | 175 |
| avrora | 83315 | 6 | MS | 106405 | 8 | 1997242 | 129 |
| | | | SS | 104579 | 8 | 139131 | 194 |
| luindex | 0 | 0 | MS | 0 | 0 | 2841482 | 745 |
| | | | SS | 0 | 0 | 539216 | 528 |
| lusearch | 0 | 0 | MS | 0 | 0 | 2823974 | 2395 |
| | | | SS | 0 | 0 | 26469692 | 2865 |
| pmd | 3 | 3 | MS | 9 | 9 | 827934 | 87 |
| | | | SS | 9 | 9 | 17378 | 189 |
| sunflow | 32 | 7 | MS | 84 | 16 | 3224426 | 123 |
| | | | SS | 64 | 16 | 65075 | 203 |
| xalan | 588 | 8 | MS | 703 | 34 | 4152344 | 149 |
| | | | SS | 704 | 36 | 203473 | 237 |

**Table 4.2:** Accuracy of LARD vs. FastTrack and a naïve low-level data-race detector. Numbers of dynamic accesses on which data-race reports occur (DRE) and distinct PCs of these reports.

We examined the differing reports and found three sources. None is inaccuracy in LARD.

First, LARD instruments all Java code in the source program, including the JDK. It therefore reports some data races involving JDK code, where applications misused unsynchronized JDK data structures. FastTrack misses these since it does not instrument the JDK. In pmd, LARD reports the same 3 data races on 3 accesses as reported by FastTrack and LARD, as well as 6 data races on unsynchronized ArrayLists from the JDK. Data races on the contents of java.util.Properties objects in xalan cause similar disparities.

Second, for each benchmark, FastTrack and LARD analyzed separate executions on entirely different JVMs, so some dynamic data-race-count variance is expected. Some benchmarks, such as sunflow, had wide variance even between runs on the same detector and JVM. There is large overlap in the data races reported. We carefully examined data races reported by only one detector. All appear possible in some executions and hidden in others, accounting for differences in sunflow and avrora. LARD reports more DREs, but at related program points.

Third, Jikes RVM is implemented for IA32 and emits two 32-bit accesses to implement accesses for Java's 64-bit long and double types. The Java Memory Model allows non-atomic long/double accesses, so we report data races individually. These duplicated reports inflate LARD's data-race counts compared to those of FastTrack, though they report the same data races, and account for the remaining disparities between LARD and FastTrack.

**Comparison with a Naïve Low-Level Data-Race Detector** The Naïve configuration reports up to several orders of magnitude more data races than LARD or FastTrack, as shown in Table 4.2. The majority of these are false data races involving system accesses; some are false data races between language-level accesses. The Naïve detector also misses some data races reported by LARD or FastTrack, as discussed in §4.4.2, and reports data races on a large number of PCs, including many outside the source program.

### 4.4.2 Impacts of LARD Extensions

To understand the practical impact of each LARD extension on missed and false data races, we compared the accurate LARD detector to variants with one or all of the LARD extensions disabled. The detectors are as follows, labeled as in Table 4.3:

- **LARD** is accurate, with all LARD extensions enabled.
- **AllMem** tracks *all* non-stack memory accesses in the JVM and the source program.
- **AllSync** tracks *all* synchronization in the JVM and the source program.
- **NoClear** ignores reports of memory deallocation.
- **NoCopy** ignores reports of memory movement.
- **Naïve** tracks all non-stack memory accesses and all synchronization and ignores all memory management.

Jikes RVM uses a one-to-one mapping between kernel threads and Java threads, so we do not evaluate a detector that ignores thread identity. For each benchmark and garbage collector

| **Bench** | **GC** | **LARD** | | **AllMem** | | **AllSync** | | **NoClear** | | **NoCopy** | | **Naïve** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **DRE** | **PC** | **F DRE** | **F PC** | **M DRE** | **M PC** | **F DRE** | **F PC** | **M DRE** | **M PC** | **F DRE** | **F PC** | **M DRE** | **M PC** |
| Java Grande | MS | 0 | 0 | 762969 | 222 | 0 | 0 | 0 | 0 | 0 | 0 | 28211 | 124 | 0 | 0 |
| | SS | 0 | 0 | 753437 | 226 | 0 | 0 | 0 | 0 | 0 | 0 | 7899 | 175 | 0 | 0 |
| avrora | MS | 106405 | 8 | 95644 | 276 | 3374 | 7 | 0 | 0 | 0 | 0 | 1894211 | 121 | 3374 | 7 |
| | SS | 104579 | 8 | 68010 | 174 | 1236 | 7 | 2 | 1 | 0 | 0 | 35788 | 186 | 1236 | 7 |
| luindex | MS | 0 | 0 | 42670 | 157 | 0 | 0 | 0 | 0 | 0 | 0 | 2841482 | 745 | 0 | 0 |
| | SS | 0 | 0 | 43460 | 159 | 0 | 0 | 0 | 0 | 0 | 0 | 539216 | 528 | 0 | 0 |
| lusearch | MS | 0 | 0 | 17079935 | 839 | 0 | 0 | 2823974 | 370 | 0 | 0 | 41760642 | 2395 | 0 | 0 |
| | SS | 0 | 0 | 16690434 | 873 | 0 | 0 | 3770587 | 264 | 0 | 0 | 26469692 | 2865 | 0 | 0 |
| pmd | MS | 9 | 9 | 43123 | 613 | 9 | 9 | 0 | 0 | 0 | 0 | 827934 | 87 | 9 | 9 |
| | SS | 9 | 9 | 17899 | 236 | 9 | 9 | 0 | 0 | 6 | 6 | 17378 | 189 | 9 | 9 |
| sunflow | MS | 84 | 16 | 99250 | 481 | 81 | 15 | 0 | 0 | 0 | 0 | 3224423 | 121 | 81 | 15 |
| | SS | 64 | 16 | 104457 | 255 | 61 | 15 | 17 | 10 | 9 | 6 | 65072 | 201 | 61 | 15 |
| xalan | MS | 703 | 34 | 837451 | 723 | 699 | 34 | 0 | 0 | 0 | 0 | 4152340 | 147 | 699 | 34 |
| | SS | 704 | 36 | 596556 | 649 | 700 | 36 | 0 | 0 | 0 | 0 | 203469 | 235 | 700 | 36 |

**Table 4.3:** False or missed data races with individual LARD extensions disabled.

(mark-sweep (MS) or semispace (SS)), we ran all six detectors *over the same execution*, comparing their results on each dynamic memory access, with LARD as ground truth (by proxy to FastTrack). Table 4.3 shows the results.

The LARD column repeats the count of racy accesses and distinct PCs for LARD from Table 4.2. For the other detectors, we report: (1) the number of false DREs (**F DRE**)—those accesses on which LARD does not report a data race but the other detector does, (2) the number of distinct PCs of false DREs (**F PC**), (3) the number of missed DREs (**M DRE**)—those accesses on which LARD reports a data race but the other detector does not, and (4) the number of distinct PCs of missed DREs (**M PC**). All four of these are reported for Naïve. For the remaining four detectors, only one of missed or false is reported. The omitted columns contain only zeroes.

The most notable feature of these results is that disabling any one extension in our experiments results in false or missed data races in at least one benchmark. *All of these LARD extensions are necessary for accuracy in practice.*

**Program vs. System** All benchmarks in our experiments have false data races under **AllMem**, with tracking of program *and* system accesses. The majority of false data races are reported on accesses outside the source program, but false data races are also reported on program accesses. *Filtering out data-race reports on PCs in system code does not eliminate all false data races.* Conversely, **AllSync** misses data races on all benchmarks, often (*e.g.*, pmd, sunflow, xalan) missing all or nearly all the data races reported by LARD.

**Memory Management** Empirically, most benchmarks do not lead to false data races when memory reuse is ignored (**NoClear**), but avrora and sunflow do suffer false data races under semispace collection. Short executions and low garbage collection pressure make problematic reuse of memory rare in these experiments. In particular, the Java Grande benchmarks generally use large, long-lived arrays, rather than short-lived shared objects. Nonetheless, false data races occurred under these relatively favorable conditions.

Ignoring memory movement in the **NoCopy** detector has no effect on the accuracy of data-race detection when using mark-sweep garbage collection, since mark-sweep never moves objects. Ignoring movement never leads to missed data races in data-race-free executions such as the Java Grande benchmarks, but missed data races do occur under semispace collection in pmd and sunflow.

### 4.4.3 Jikes LardVM Performance on Lardish

We did a preliminary evaluation of the performance potential of hardware-based Lard data-race detection for Java by running benchmarks from DaCapo Jikes LardVM on simulated Lardish hardware. We model the same baseline configuration as in [38], extended with mechanisms described in §4.3.2. A PIN [74] binary instrumentation tool emulates Lardx86 to drive the simulator, an extension of the simulator used in [38]. Our main addition is to model the costs of ClearHistory and CopyHistory. For each, we charge 100 cycles to transition to the software manager. Next we simulate memory accesses for software manager lookups of what access histories may need to be invalidated in the cache, issuing invalidations (the same cost as a cache hit for the cache where the access history lives in the cache hierarchy). For CopyHistory, we simulate the accesses of the software handler for eviction; in ClearHistory there is no handler for eviction since the data is discarded. Finally, we simulate each memory access required in the software manager to clear or copy software access histories.

Results from these initial simulations suggest Jikes LardVM on Lardish has overheads under 50% in most cases, comparable to those reported for C programs in [38], while lower on average. JVM accesses are not data-race-checked, so Lardish can introduce less overhead than Radish does for C programs in which all accesses are checked.

### 4.4.4 Jikes LardVM Performance on x86

We evaluated the performance overhead of our JVM modifications alone on real x86 hardware (with compilers emitting no Lardx86 instructions) to measure the costs of tracking synchronization and memory management in software. We ran benchmarks from DaCapo with their largest input 10 times each with Jikes LardVM and with an unmodified Jikes RVM, and with the mark-sweep and semispace collectors. Figure 4.3 shows the additional overheads of Jikes LardVM normalized to unmodified Jikes RVM using the same garbage

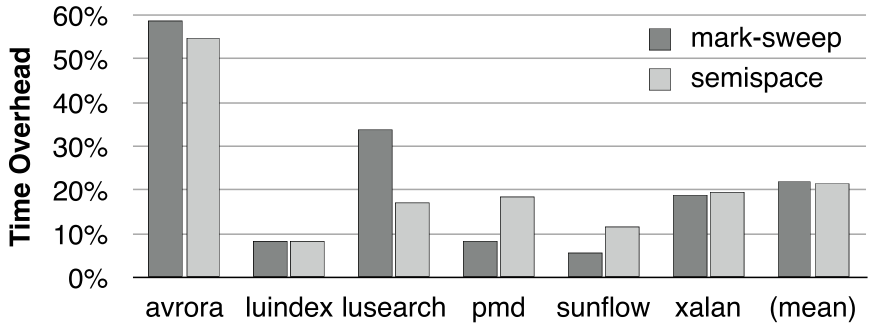


**Figure 4.3:** Execution time overhead of Jikes LARDVM normalized to unmodified Jikes RVM, both run on native x86.

collector. Overheads average 22% for mark-sweep and 21% for semispace. All overheads are under 60% and most are under 20%. These results show that synchronization and memory management events can be tracked in software with relatively low overhead, even in our relatively unoptimized prototype.

## 4.5 Related Work

Work related to low-level abstractable race detection generally falls into two categories: work on virtualizing low-level resources or providing usable semantics when compiling under-specified program primitives (§4.5.1) and work on data-race detection that has attempted to address one or more of the translation issues that affect low-level data-race detection for high-level languages (§4.5.2).

### 4.5.1 Virtualization and Language Semantics

The task of virtualizing data-race detection bears some resemblance to virtualization of other hardware services, such as virtual transactional memory [107]. However, unlike hardware transactional memory, for example, data-race detection is not an execution resource to be controlled by programs. It is more an issue of preserving semantics.

A better parallel is Boehm's exposition of why a usable semantics for shared-memory multithreading requires deep language integration, hence threads cannot be implemented as a library [18]. This work served as a precursor to the formal C/C++ memory model [20], which addresses issues raised in [18]. Just as compilers must reason carefully about the semantics of synchronization and memory accesses to translate shared-memory multithreaded programs to machine code in a way that preserves usable semantics, they must also faithfully translate the language-level notion of a data race to a machine-level analysis if the analysis results are to be interpreted at the language level. Our approach has also been similar to that in [18], starting with practical correctness problems in low-level detection of language-level data races and deriving a more general set of rules to solve them. The responsibilities of a compiler in low-level abstractable race detection are analogous to the compiler's responsibilities under a language memory consistency model [20, 70, 76].

Systems that try to use low-level analysis to reason about the effects of data races as harmful or benign [65, 87] have been limited by reasoning about a specific low-level instantiation of a higher-level program in a language with a permissive semantics [19]. Like a naïve low-level data-race detector, they lack sufficient semantic information from the language-level program. A system in the spirit of LARD might help these analyses become more relevant at the language level, although their judgments on effects of data races have farther-reaching dependence on language semantics than merely detecting data races.

### 4.5.2 Compensation for Translation Artifacts

Others have previously identified memory management or custom synchronization as potential sources of inaccuracy when using low-level data-race detection implementations for high-level languages, but existing solutions do not fully address this inaccuracy.

Helgrind [125] offers C preprocessor macros to annotate custom synchronization and allocation in C/C++ programs to reduce false data-race reports, but there is no way to turn off analysis of standard synchronization operations (*e.g.*, pthreads) or to specify memory movement in a way that preserves data-race detection access history. RADISH (Chapter 3) depends on synchronization libraries to annotate synchronization effects for the data-race detector, but does not provide support for memory reuse or movement. LARD provides a general interface to communicate the salient details of language-level synchronization and memory abstractions to the low-level data-race detector for accurate data-race detection.

Choi, *et al.*, mention that memory reuse and movement in garbage collection could break their JVM-level data-race detector, since it stores addresses of objects in its analysis state [30]. Their solution (a heap size large enough that garbage collection never occurs) is not feasible for production systems. Qi, *et al.*, address this problem in their MulticoreSDK [105] JVM data-race detector implementation with logical addresses for data-race-checked locations. This adds a logical address lookup indirection to the critical path of memory accesses, along with the cost of managing available unique identifiers. For low-level data-race detectors, each tracked memory instruction in the ISA would need to take an extra logical-address argument. TLB-like hardware support to cache the low-level to logical address mapping could speed the lookup at the cost of lengthening the critical path for cache accesses, which is unmodified in hardware data-race detectors such as RADISH (Chapter 3), but would require shootdown on object movement. LARD uses explicit movement of analysis state to follow the movement of objects, which occurs much less frequently than memory accesses. While reporting movement has overheads to copy access histories, data-race-checked memory accesses occur *far* more frequently than object movement. We choose reporting movement (copying) as a less invasive and higher-performance option.

RACEZ [118] is an offline, dynamic data-race detector for C/C++ that uses the imprecise lockset algorithm [115], sampling of memory accesses via hardware performance monitoring, and offline log analysis to improve run-time performance at the cost of missing some true data races and reporting some false data races. RACEZ uses memory allocation events as a *heuristic* to filter false positives, but the authors give no discussion of why tracking memory allocation is important—-nor to what extent it is effective—in reducing false positives. Memory movement is not considered.

This prior work indicates that memory management affects the accuracy of data-race detection, but our work is the first to provide a single clear interface for managing all aspects of the gap between language-level and low-level memory abstractions and to make low-level data-race detection implementations accurate for language-level data-race detection.

## 4.6 Conclusions

LARD is a set of extensions to low-level data-race detectors and run-time systems. LARD's five extensions—distinguishing program vs. system memory accesses, distinguishing program vs. system synchronization, reporting memory reuse, reporting memory movement, and reporting thread identity—are sufficient for low-level detection of language-level data races. We have implemented a prototype system to detect data races in Java programs using a low-level data-race detector and modifications to a JVM. Our results demonstrate that basic low-level detection mechanisms alone do not provide accurate detection of language-level data races, but a LARD implementation does. LARD admits general hardware implementations of performance-critical data-race checking logic, while allowing language implementations to customize the semantics of memory, synchronization, and thread identity. This result is a step toward feasible data-race exceptions for high-level languages.

Chapter 5

# FIB: Fast Instrumentation Bias

*Fast Instrumentation Bias (FIB)* aims to reduce the cost of data-race checks in pure-software dynamic data-race detectors by using cooperative synchronization to protect against *metadata races* that could compromise accuracy.

## 5.1 Introduction

*Who detects the races in the race detector?* Software dynamic analysis implementations modify target programs by inserting analysis code inline within application code and running the modified program directly. The execution of the analysis is therefore subject to all effects of program execution. In addition to analyzing concurrency properties of the program execution, a data-race detector (or other similar analysis implementation) must protect itself against consistency errors due to *metadata races* induced by the very concurrency patterns the analysis intends to detect.

A software dynamic data-race detector inserts code called a *barrier* before each memory access in the program to check whether that access will cause a data race. This barrier reads and may update analysis metadata that is stored in memory alongside program data. When two program accesses to the same memory location execute concurrently with each other, their associated barriers also execute concurrently. Without sufficient additional synchronization in analysis barriers, metadata races and atomicity violations in these barriers may corrupt the analysis metadata. Ironically, this is most likely to occur exactly when the analysis should be detecting a data race in the program, and when it causes the analysis to

| **Thread** $t_1$ | **Thread** $t_2$ | $W_x$ | $R_x$ |
|---|---|---|---|
| $C_{t_1} = \{5@t_1, 3@t_2, 1@t_3\}$ | $C_{t_2} = \{2@t_1, 7@t_2, 1@t_3\}$ | | |
| | | $1@t_3$ | $1@t_3$ |
| Load $1@t_3$ from $W_x$ | | | |
| Is $1@t_3 = C_{t_1}(t_1)$? No. | Load $1@t_3$ from $R_x$. | | |
| Load $1@t_3$ from $R_x$. | Is $1@t_3 = C_{t_2}(t_2)$? No. | | |
| | Is $1@t_3 \preceq C_{t_2}$? Yes, OK! | | |
| Is $1@t_3 \preceq C_{t_1}$? Yes, OK! | Store $C_{t_2}(t_2)$ in $R_x$. | | $7@t_2$ |
| Store $C_{t_1}(t_1)$ in $W_x$. | | $5@t_1$ | |
| Store $C_{t_1}(t_1)$ in $R_x$. | | | $5@t_1$ |
| | rd$(t_2, x, v_2)$ | | |
| wr$(t_1, x, v_1)$ | | | |

**Figure 5.1:** Lost access history updates lead to a missed data race in an unsynchronized software data-race detector. Time flows down. Thread $t_1$'s Load from $R_x$ should observe the value written by thread $t_2$'s Store in $R_x$ to detect the data race, but does not in this interleaving. Other safe and unsafe orderings are also possible.

miss a true data race in the program execution.

**Example** Consider Figure 5.1, in which two FastTrack barriers[1] and the accesses they analyze execute concurrently without synchronization, resulting in a lost access history update and a missed data race.

Thread $t_1$ prepares for a write to data location $x$ by executing a write barrier. It first checks if the last write occurred in the same epoch, which it does not. The barrier next loads the last read and checks if it is by this thread. The barrier then checks if thread $t_1$'s current epoch happens after the last read, finding that it does. This shows that all accesses completed so far happen before the upcoming write in thread $t_1$, so there is no data race. Thread $t_1$ therefore updates the access history to record its current epoch as the last write and last read.

However, thread $t_2$ is executing a read barrier on $x$ concurrently with thread $t$'s write barrier. Thread $t_2$'s read barrier also checks against the last read for ordering, but it observes

[1]Refer to Figures 2 and 5 in [52] for presentation of the original FastTrack barriers used in Figure 5.1 of this dissertation. Additionally §5.2 in this dissertation presents the slightly modified version of FastTrack we use in the remainder of this chapter.

the last read before thread $t_1$ has updated it, finding that the last read happens before thread $t_2$'s current epoch, so it proceeds to record a new last read. There is clearly a data race between the data write in thread $t_1$ and the data read in thread $t_2$, but neither barrier has detected this, violating the guarantee that one or both of the pair of racing accesses will raise a data-race exception.

Note that even if the read barrier also checked against the last write, lost updates and missed races could still occur.

Some data-race detectors simply forego formal accuracy guarantees in order to omit metadata synchronization altogether, since the timing of concurrent barriers must be simultaneous in a very small window to compromise analysis accuracy in practice. This often suffices for best-effort debugging tools, but data-race exceptions require strong accuracy guarantees.

### 5.1.1 Barrier Atomicity and Barrier-Access Ordering

A sufficient solution to the metadata consistency problem is to ensure every barrier-access sequence executes atomically. *Barrier-access atomicity* ensures that the analysis of an access always executes on metadata that is consistent with the state of the target program execution. The operational semantics for FastTrack presented in [52] has this feature.

In practice, *barrier-access atomicity* is stronger than necessary to implement the *first-race* guarantee (§2.2.3). If the analysis never allows execution to pass a barrier that detects a data race, then the weaker policy of *barrier atomicity* and *barrier-access ordering* suffices to ensure analysis consistency.

- *Barrier atomicity:* All barriers that do not detect data races execute atomically. If a barrier's checks determine an access to be data-race-free, updates to the access history to record the new access occur atomically with the metadata observations on which the checks were based. If a barrier detects a data race, it may execute non-atomically, but rolls back any access history updates it has made and raises an exception.

- *Barrier-access ordering:* Every barrier completes and makes any access history updates visible in memory *before* the access it monitors, and in the same synchronization-free region. A compiler may even reorder barriers and corresponding accesses individually, so long as it respects the original barrier-access order and does not move any program synchronization operations between an access and its barrier.

On hardware memory models that do not relax read-read or read-write order across control dependences, no fence is necessary to enforce ordering between the data-race check and the update. Thus no fence is necessary for any implementation of barrier atomicity under SC, TSO, PC, or PSO [3]. Otherwise, on weaker memory models that relax read-read or read-write order, we must prevent program writes from becoming visible and program reads from being used before the barrier has successfully shown data-race freedom. It would seem unsafe for a program access to take effect before its barrier update in the general case, but this can cause the data-race detector to miss a true data race only if it also violates barrier atomicity. The need for a fence to enforce ordering of the data-race check and the program access depends on the implementation of barrier atomicity. We suspect that most, if not all, reasonable and correct implementations of barrier atomicity would also suffice to force the proper ordering. Our design and implementation targets TSO, and for simplicity we assume TSO in the remaining discussion.

Given these constraints and the symmetry of the data-race detection algorithm (a pair of barriers on racing data accesses will detect the data race regardless of which barrier executes first), both analysis barriers for a pair of racing data accesses are guaranteed to execute—and detect the data race—before at least one of the racing data accesses executes. Thus arbitrary interleaving of barriers and corresponding accesses to the same memory location from multiple threads can never lead to false or missed data races, given barrier atomicity and barrier-access ordering.

### 5.1.2 Pessimistic Barrier Atomicity

Barrier-access ordering is fairly straightforward and inexpensive to implement, but barrier atomicity can be costly in software. Barrier atomicity is feasible with little or no additional overhead in a hardware-supported data-race detector, as in RADISH (Chapter 3). A software dynamic data-race detector stores its analysis metadata in shared memory, so synchronization is necessary to ensure barrier atomicity in the presence of concurrent accesses to the same memory location, whether or not they are data races.

An obvious software implementation of barrier atomicity is to protect the barrier with a critical section on a lock associated with the accessed memory location, so that analysis metadata for the memory location is accessed by at most one barrier at a time. This pessimistic solution can incur significant overhead in practice. Profiling experiments in [38] suggest that pessimistic barrier atomicity accounts for 20%-90% of the additional overhead of the original FastTrack implementation [52] beyond an unmodified HotSpot JVM running four multithreaded benchmarks from the Java Grande suite [122]. Measurement of the DaCapo benchmarks [14] on our own separate implementation of FastTrack in Jikes RVM shows that even a mostly-lock-free implementation of barrier atomicity accounts for 3-38% of the *full execution time* of the DaCapo benchmarks.[2] Costs for a naïve barrier atomicity implementation acquiring a spin lock for every barrier are even higher. Thus eliminating the costs of barrier atomicity has the potential for significant overall performance improvement.

### 5.1.3 Cooperative Barrier Atomicity with Fib

We propose the *Fast Instrumentation Bias (Fib)* protocol for cooperative synchronization based on per-location thread ownership state. Fib redistributes the synchronization burden of barrier atomicity such that the expected common case requires no synchronization, while rare cases require costlier synchronization. As a cooperative synchronization protocol, Fib is inspired by insights from biased locking [67, 99, 113], cache coherence [96], local permissions in Radish (§3.2.3.2), and cooperative object-granular thread conflict analysis in Octet [23]. These protocols, discussed further in §5.7, track and control concurrent resource usage with thread-ownership states, generally including at least a state where access is available exclusively to thread $t$ and a state where read-only access is shared by a set of threads.

The principal difference with pessimistic access control is that when a thread, $t$, needs

[2]These percentages are not directly comparable across sources, even ignoring the different implementations and benchmarks. The experiments in [38] report barrier atomicity costs as a percentage of the execution time added to baseline JVM execution time by data-race detection. The experiments in this work report how much of the full execution time, including baseline JVM execution time *and* the additional execution time due to data-race detection. The former percentages would be smaller under the latter scheme; the latter percentages would be larger under the former scheme.

to access a resource currently owned exclusively by another thread, $u$, thread $t$ acquires ownership not by synchronization through shared memory in that resource (*e.g.*, by acquiring a lock attached to the access history), but by direct communication with thread $u$. The owner thread, $u$ responds to this request explicitly and only at well-defined points in the program. A thread can observe that it holds exclusive or shared ownership of a resource without any synchronization or cross-thread communication. This thread then enjoys the guarantee that the ownership state will not change until this thread explicitly changes it in response to another thread's request.

FIB is unique among cooperative synchronization protocols because it is specific to data-race detection and it uses no dedicated storage to represent ownership state. Instead, it derives ownership state purely from access history metadata in the data-race detector. Transitions from one state to another are tied to access history updates in the data-race detection algorithm.

### 5.1.4 Contributions and Outline

*Fast Instrumentation Bias (FIB)* aims to reduce the cost of data-race checks in pure-software dynamic data-race detectors by using cooperative synchronization to protect metadata.

**Hypothesis:** Performance overheads of pure-software accurate dynamic data-race detection can be reduced by replacing pessimistic metadata synchronization with *instrumentation bias*, a form of cooperative metadata synchronization based on thread ownership information already implicitly encoded by data-race detection metadata.

Our contributions in the remainder of this chapter are organized as follows:

- We present a simple accuracy-preserving modification to the original FastTrack algorithm that we will use as the basis for the remainder of Chapter 5. (§5.2)

- We present *Fast Instrumentation Bias (FIB)*, an algorithm for accurate data-race detection that uses cooperative synchronization to control access to access history with no synchronization in common cases in exchange for costly synchronization in rare cases. (§5.3)

- We apply conservative dynamic thread-escape analysis as a pre-filter for data-race detection and show how it maintains accuracy. (§5.4)

- We implement FIB for Java programs in the Jikes RVM [8] Java virtual machine. (§5.5)

- We evaluate the overall performance of multithreaded Java applications on our prototype implementation as well as unsynchronized and pessimistically synchronized implementations of FastTrack, finding that FIB is 13-21% faster than the fastest conventionally synchronized implementation on 4 benchmarks, 0-7% slower on three benchmarks, and 45-260% slower on three benchmarks, and that dynamic thread-escape filtering has mixed performance effects. We profile the distribution of intended common and rare cases in practice, finding that poor performance in FIB is linked to relatively high rates of FIB's expensive slow paths, from roughly 0.4% to 2.2% of barriers. (§5.6)

- We discuss related work on cooperative synchronization and dynamic thread-escape analysis. (§5.7)

- We discuss limitations of FIB as described in this dissertation. (§5.8)

- We propose future extensions to address FIB's limitations and improve performance more generally. (§5.9)

- Finally, we conclude (§5.10).

## 5.2 FastTrack

FIB is derived from the state-of-the-art FastTrack algorithm for accurate dynamic data-race detection [52], also introduced briefly in §2.2.1.3. This section describes a slightly modified version of FastTrack that is the basis for FIB (§5.3) and all data-race detector implementations described in §5.5 and evaluated in §5.6. Our modification—storing the epoch of the last write as the last read if there is no last read since the last write—makes some checks cheaper and supports FIB's derivation of ownership state from access history. We discuss the essence of the modification in §5.2.2 and note its implications throughout this section.

An *epoch*, $e = c@t$, is a local logical time, $c$, in a single thread, $t$. We use *vector clocks*, $\mathrm{v}$, interchangeably with sets of epochs with at most one belonging to a given thread, such that a vector clock lookup returns an epoch: $\mathrm{v}(t) = c@t$.[3] Epoch $c@t$ happens before vector clock $\mathrm{v}$, written $c@t \preceq \mathrm{v}$, if and only if $\mathrm{v}(t) = d@t$ and $c \leq d$. FastTrack uses the same synchronization tracking as the canonical vector-clock algorithm (§2.2.1.1). The origin epoch, $0@t_0$, is an epoch that happens before all logical times of operations of all threads. It is notated $\perp_{\mathsf{e}}$ in [52].

### 5.2.1 Access History

The FastTrack access history for location $x$ differs slightly from that of the canonical vector-clock algorithm. Our modified version stores the following:

- **Last Write** $W_x$**:** the epoch of the last write to $x$. If $x$ has never been written, the last write is considered to have happened in the origin epoch, $0@t_0$.

- **Last Read(s)** $R_x$**:**
  - If there has been at least one read since the last write and all reads since the last write are totally ordered by happens-before, then the value is the epoch of the last read of $x$.
  - If there have been mutually concurrent reads since the last write, then the value is a set (or *read map*) $\mathrm{v}$ where each element $c@t$ is the epoch of thread $t$'s last read from $x$ since the last write. This set is typically represented by a vector clock or other *read map* structure.
  - If no reads have occurred since the last write, then the value is the epoch of the last write, if any, or the origin epoch, $0@t_0$, otherwise.

[3]This is a minor change from the vector clocks introduced in §2.2.1, which store raw clocks, not epochs.

In this case only, the version of FastTrack used in this chapter differs from the original. In the original FastTrack algorithm, the value of $R_x$ may also be an epoch that happens before the last write epoch.

### 5.2.2 Invariants

Our version of FastTrack maintains the following invariants for each access history, making it sound for data-race checks to examine only the partial access history described above.

**Invariant 1** *All writes to $x$ in the execution so far and all reads that precede the last write recorded as $W_x$ in the execution* happen before *epoch $W_x$ or happen in epoch $W_x$.*

This invariant is also maintained by the original FastTrack algorithm.

**Invariant 2** *If $R_x$ is a set* v*, then all accesses to $x$ in the execution so far* happen before *at least one epoch $e \in r$.*

This invariant is also maintained—but its full strength is not exploited—by the original FastTrack algorithm.

**Invariant 3** *If $R_x$ is a single epoch, $e$, then all accesses to $x$ in the execution so far* happen before *epoch $e$ or happen in epoch $e$.*

This invariant is *not* maintained by the original FastTrack algorithm. By changing the original semantics of $R_x$ to *the epoch(s) of the most recent accesses to $x$ in happens-before order*, we gain Invariant 3. With this invariant, any check that can show $R_x$ happens before the current logical time need not check against $W_x$, as discussed in §5.2.3.1 and §5.2.3.2. More importantly, it affords Fib a simple derivation of ownership state by examining $R_x$ alone, as described later in §5.3. While $R_x$ would now more accurately be named the *last access*, we retain the name *last read* for consistency with existing terminology.

### 5.2.3 Barriers

Figure 5.3 shows our FastTrack barriers as a judgment $(C; L; R; W) \overset{a}{\Longrightarrow}_{\mathsf{FT'}} (C'; L'; R'; W')$ on the data-race detector state, shown in Figure 5.2, and a program operation. We omit

$$\begin{array}{rccl}
\text{Clock} & c & \in & \mathbb{N} \\
\text{Epoch} & e & ::= & c@t \\
\text{Vector Clock} & \mathrm{v} & ::= & \cdot \mid \mathrm{v}, t \mapsto c@t \\
\text{Thread VCs} & C & ::= & \cdot \mid C, t \mapsto \mathrm{v} \\
\text{Lock VCs} & L & ::= & \cdot \mid L, m \mapsto \mathrm{v} \\
\text{Last Reads} & R & ::= & \cdot \mid R, x \mapsto e \mid R, x \mapsto \mathrm{v} \\
\text{Last Writes} & W & ::= & \cdot \mid W, x \mapsto e \\
\text{FastTrack State} & (C; L; R; W) & &
\end{array}$$

**Figure 5.2:** FastTrack metadata

synchronization tracking, which is identical to that of the canonical vector-clock algorithm, described in §2.2.1.1 and Figure 2.3. Program and heap constraints are imposed by an external judgment (not shown here) that uses this judgment for data-race detection. Stuckness indicates a data race. This high-level semantics has barrier atomicity by virtue of the atomicity of a single step, but does not indicate how it might be implemented in practice.

#### 5.2.3.1 Write Barrier

A **write** to location $x$ by thread $t$ is data-race-free if all accesses to $x$ in the execution so far happen before thread $t$'s current logical time.

If the last write is the current epoch (FT' WRITE EXCL SAME EPOCH: $W_x = C_t(t)$), then a write in this epoch has already been shown to be data-race-free with earlier accesses and later accesses will check for data races with writes in this epoch. Further checks or updates are redundant. This is FastTrack's FT WRITE SAME EPOCH case [52].

Otherwise, the write barrier must check ordering with the last reads. (Due to Invariant 3 it is never necessary to explicitly check against the last write in our modified version.)

If $R_x$ is an epoch, $e$, then:

- FT' WRITE EXCL SAME READ THREAD: If $e$ is an epoch of this thread, then the last read happens before the current access by program order. By transitivity via Invariant 3, all previous accesses happen before the current access.

$$\boxed{(C; L; R; W) \xRightarrow{a}_{\mathsf{FT'}} (C'; L; R'; W')}$$

FT' READ EXCL SAME EPOCH

$$\frac{R_x = C_t(t)}{(C; L; R; W) \xRightarrow{\mathsf{rd}(t,x,v)}_{\mathsf{FT'}} (C; L; R; W)}$$

FT' READ EXCL SAME THREAD

$$\frac{R_x = c@t}{(C; L; R; W) \xRightarrow{\mathsf{rd}(t,x,v)}_{\mathsf{FT'}} (C; L; R, x \mapsto C_t(t); W)}$$

FT' READ EXCL

$$\frac{R_x = e \qquad e \preceq C_t}{(C; L; R; W) \xRightarrow{\mathsf{rd}(t,x,v)}_{\mathsf{FT'}} (C; L; R, x \mapsto C_t(t); W)}$$

FT' READ SHARE

$$\frac{R_x = c@u \qquad W_x \preceq C_t \qquad \mathrm{v} = \cdot, u \mapsto c@u, t \mapsto C_t(t)}{(C; L; R; W) \xRightarrow{\mathsf{rd}(t,x,v)}_{\mathsf{FT'}} (C; L; R, x \mapsto \mathrm{v}; W)}$$

FT' READ SHARED SAME EPOCH

$$\frac{R_x = \mathrm{v} \qquad \mathrm{v}(t) = C_t(t)}{(C; L; R; W) \xRightarrow{\mathsf{rd}(t,x,v)}_{\mathsf{FT'}} (C; L; R; W)}$$

FT' READ SHARED AGAIN

$$\frac{R_x = \mathrm{v} \qquad t \in \mathrm{v} \qquad \mathrm{v}' = \mathrm{v}, t \mapsto C_t(t)}{(C; L; R; W) \xRightarrow{\mathsf{rd}(t,x,v)}_{\mathsf{FT'}} (C; L; R, x \mapsto \mathrm{v}'; W)}$$

FT' READ SHARED FIRST

$$\frac{R_x = \mathrm{v} \qquad W_x \preceq C_t \qquad \mathrm{v}' = \mathrm{v}, t \mapsto C_t(t)}{(C; L; R; W) \xRightarrow{\mathsf{rd}(t,x,v)}_{\mathsf{FT'}} (C; L; R, x \mapsto \mathrm{v}'; W)}$$

FT' WRITE SAME EPOCH

$$\frac{W_x = C_t(t)}{(C; L; R; W) \xRightarrow{\mathsf{wr}(t,x,v)}_{\mathsf{FT'}} (C; L; R; W)}$$

FT' WRITE EXCL SAME READ EPOCH

$$\frac{R_x = C_t(t)}{(C; L; R; W) \xRightarrow{\mathsf{wr}(t,x,v)}_{\mathsf{FT'}} (C; L; R; W, x \mapsto C_t(t))}$$

FT' WRITE EXCL SAME READ THREAD

$$\frac{R_x = c@t}{(C; L; R; W) \xRightarrow{\mathsf{wr}(t,x,v)}_{\mathsf{FT'}} (C; L; R, x \mapsto C_t(t); W, x \mapsto C_t(t))}$$

FT' WRITE EXCL

$$\frac{R_x = e \qquad e \preceq C_t}{(C; L; R; W) \xRightarrow{\mathsf{wr}(t,x,v)}_{\mathsf{FT'}} (C; L; R, x \mapsto C_t(t); W, x \mapsto C_t(t))}$$

FT' WRITE SHARED

$$\frac{R_x = \mathrm{v} \qquad \mathrm{v} \sqsubseteq C_t}{(C; L; R; W) \xRightarrow{\mathsf{wr}(t,x,v)}_{\mathsf{FT'}} (C; L; R, x \mapsto C_t(t); W, x \mapsto C_t(t))}$$

**Figure 5.3:** High-level view of modified FastTrack barriers.

- FT' WRITE EXCL: If $R_x \preceq C_t$, then by Invariant 3, all previous accesses happen before the current epoch. Both the last write and last read must be set to the current epoch to maintain Invariant 3. This corresponds to FastTrack's FT WRITE EXCLUSIVE case, which also checks against $W_x$, but leaves $R_x$ unchanged.

- Otherwise, this write forms a data race with at least the last read.

If $R_x$ is a read map, v, then:

- FT' WRITE SHARED: If $R_x \sqsubseteq C_t$, then by Invariant 2, all previous accesses happen before the current epoch. Both the last write and last read must be set to the current epoch to maintain Invariant 3. This corresponds to FastTrack's FT WRITE SHARED case, which additionally checks against $W_x$ and sets $R_x = 0@t_0$.

- Otherwise, this write forms a data race with at least one of the last reads.

#### 5.2.3.2 Read Barrier

A **read** to location $x$ by thread $t$ is data-race-free if all writes to $x$ in the execution so far happen before the current epoch. There are multiple fast paths in the read check.

If $R_x$ is an epoch, $e$, then:

- FT' READ EXCL SAME EPOCH: If $e$ is the current epoch ($C_t(t)$), then a read in this epoch has previously been found to be data-race-free and has been recorded. Checking and recording again would be redundant. This is FastTrack's FT READ SAME EPOCH case [52].

- FT' READ EXCL SAME THREAD: Otherwise, if $e$ is an epoch of thread $t$, but is not the current epoch, then by program order it happens before the current epoch. By the transitive happens-before invariants (§5.2.2) all previous accesses also happen before the current epoch. The last read must be set to the current epoch. The original FastTrack algorithm does not distinguish this case from the next, more general, case.

- FT' READ EXCL: Otherwise, if $e \preceq C_t$, then all earlier accesses happen before this access. The last read must be set to the current epoch. By our modification's introduction of Invariant 3, it is never necessary to check against the last write in this case as the original FastTrack algorithm does in its corresponding FT READ EXCLUSIVE case [52].

- FT' READ EXCL: Otherwise, if $e \not\preceq C_t$, but $W_x \preceq C_t$, then this read is data-race-free with earlier writes, but concurrent with some other read to $x$. In this case, the last read must be set to a new read map containing the previous single last read epoch and the current epoch. This is FastTrack's FT READ SHARE case [52].

- Otherwise, this access forms a data race with at least the last write.

If $R_x$ is a read map, v, then:

- FT' READ SHARED SAME EPOCH: If thread $t$ has an entry in v and it is the current epoch, then no further checks or updates are needed, as in the corresponding single-epoch case above.

- FT' READ SHARED AGAIN: Otherwise, if thread $t$ has an entry in v, then thread $t$ has already read since the last write. By program order and the happens-before order established by that earlier check, the last write happens before the current epoch, and it is not necessary to explicitly check $W_x \preceq C_t$. The last read entry for $t$ must be set to the current epoch.

- FT' READ SHARED FIRST: Otherwise, thread $t$ has no entry in v and has not read $x$ since the last write and there is no transitive ordering via program order. If $W_x \preceq C_t$, then the last write happens before the current epoch. The last read entry for $t$ must be set to the current epoch. This is FastTrack's general FT READ SHARED case, which covers the above two cases as well [52].

- Otherwise, this read forms a data race with at least the last write.

## 5.3 The Fib Protocol

In this section, we describe *Fast Instrumentation Bias (Fib)*, a dynamic data-race detection algorithm which extends the FastTrack algorithm (as modified in §5.2) with a protocol for cooperative synchronization of data-race detection metadata. Fib's logic for checking for data races is nearly identical to the FastTrack algorithm, but this analysis logic is intertwined with logic to maintain the consistency of the data-race detection metadata without pessimistic synchronization.

Fib derives an ownership state for each memory location purely from that location's access history, with no dedicated storage of the ownership state. In each analysis barrier, Fib uses this ownership state to determine what (if any) synchronization is required to ensure barrier atomicity. The first step of a data-race check subsumes the work to determine the current ownership state of the access history this check will use. In the common case, the joint check shows that this thread has exclusive or shared ownership of the location and that there is no data race. Thus the data-race check and access history update execute without any synchronization. In the rare case, when the ownership state does not grant this thread sufficient permission, cross-thread coordination is required to preserve barrier atomicity.

### 5.3.1 Notation

Fib uses the per-memory-location, per-thread, and per-synchronization-object analysis metadata of our modified FastTrack (§5.2) and adds some thread metadata to support cross-thread communication. Throughout this section, analysis pseudo-code uses records describing thread metadata and access history metadata. Figure 5.4 shows the form of these records and their correspondence to FastTrack metadata described in §5.2. Dot notation denotes record field access. Synchronization metadata and tracking are identical to FastTrack; we omit them here.

Every access history is allocated with its `lastReads` field initialized to the single epoch $0@t$ where $t$ is the allocating thread. This epoch happens before all operations of all threads in the program (for all threads $t$, thread $t$'s initial vector clock holds the epoch 0 for all threads other than $t$), so it does not affect the results of data-race checks. We discuss options

```
Thread {                                AccessHistory {
  VC vc; ← $C_t$                          Epoch      lastWrite; ← $W_x$
  RequestQueue queue;                     Epoch ∪ VC lastReads; ← $R_x$
  AckQueue     acks;                    }
  Response     response;
}
```

**Figure 5.4:** FIB analysis metadata. The type notation `Epoch ∪ VC` indicates that the `lastReads` field holds either an epoch or a reference to a read map, represented as a vector clock.

and optimizations for ownership initialization in §5.4.2.

In the following sections, FIB barriers and communication mechanisms are described with Java-like pseudo-code to be executed with total store order (TSO, [3]) memory consistency semantics. Stores are retired in program order; store-load order can be enforced by issuing a memory fence with `fence()`. In barriers and supporting routines, types are elided and local variables are scoped at the function level. Variables and parameters `t` and `u` are references to `Thread` records. Variables `h` or `history` are references to `AccessHistory` records. Barriers take as arguments the currently executing thread and the access history of the data location to be accessed. A value $r$ from the `lastReads` field of an access history can be distinguished as an epoch or a vector clock with the functions `isExclusive(`$r$`)` or `isShared(`$r$`)`, respectively. The thread of an epoch $e$ is extracted with `thread(`$e$`)`. The order of two epochs of the same thread can be compared with $>$, $<$, $\leq$, and $\geq$. For example, $c@t \leq d@t$ is equivalent to $c \leq d$.

## 5.3.2 Ownership States

The use of an access history is controlled by an ownership state derived from the access history. Each ownership state grants certain threads permission to perform certain operations on the access history without synchronization, along with the guarantee that the ownership state will not change without cooperation from these threads. There are two types of ownership states, with corresponding permissions:

- $\mathsf{Exclusive}(t)$ grants thread $t$ exclusive permission for both write and read barriers to check against and update all contents of this access history without synchronization.

- $\mathsf{Shared}(S)$ grants every thread $t \in S$ shared permission for read barriers to check against all contents of this access history and update thread $t$'s entry in the access history's last reads map without synchronization.

There is no dedicated storage for the ownership state of an access history. The read component of an access history determines ownership:

- $\mathsf{Exclusive}(t)$: If the `lastReads` field holds a non-zero epoch, $c@t$ where $c > 0$, then thread $t$ has *exclusive ownership* of the access history.

- $\mathsf{Shared}(S)$: If the `lastReads` field holds a read map $r$, then all threads $t$ with an entry in $r$ hold *shared ownership* of the access history: $\mathsf{Shared}(S)$, where $S = \{t \mid r[t] = c@t \wedge c > 0\}$.

All barriers that are not granted sufficient ownership under the current state must initiate a state transition, which involves coordination (via cross-thread communication) with all threads that do hold ownership. The one exception is read barriers by threads not in the shared set: all threads $u \notin S$ may gain shared ownership via a communication-free, almost-as-fast path. State transitions are described in §5.3.3; the special read-shared case is discussed in §5.3.4.2.

Since access histories for data allocated by thread $t$ have their last read initialized to the single epoch $0@t$, the initial ownership state is naturally $\mathsf{Exclusive}(t)$. We discuss options and optimizations for ownership initialization in §5.4.2.

When the specific threads involved are not important, an exclusive state $\mathsf{Exclusive}(t)$ may be abbreviated as $\mathsf{Exclusive}$ and a shared state $\mathsf{Shared}(S)$ may be abbreviated as $\mathsf{Shared}$.

Note that while a $\mathsf{Shared}$ state corresponds to the conventional notion of *read-shared* data, the access history may receive updates: read barriers by any thread $t$ may update its entry in the last reads map. However, these updates are always non-conflicting (each thread

| Profile | Type | Barrier | Start State | → | End State | Coordination | DRE? |
|---|---|---|---|---|---|---|---|
| fast, common | local | write | Exclusive($t$) | $\rightarrow$ | Exclusive($t$) | none | no |
| | | read | Exclusive($t$) | $\rightarrow$ | Exclusive($t$) | | |
| | | | Shared($S$), $t \in S$ | $\rightarrow$ | Shared($S$) | | |
| medium, rare[a] | fence | read | Shared($S$), $t \notin S$ | $\rightarrow$ | Shared($S \cup \{t\}$) | fence | maybe |
| slow, rare | single-conflict[b] | write | Exclusive($u$), $u \neq t$ | $\rightarrow$ | Exclusive($t$) | comm. $u$ | |
| | | read | Exclusive($u$), $u \neq t$ | $\rightarrow$ | Exclusive($t$) | | |
| | | | Exclusive($u$), $u \neq t$ | $\rightarrow$ | Shared($\{u, t\}$) | | |
| slower, rarer | multiple-conflict | write | Shared($S$) | $\rightarrow$ | Exclusive($t$) | CAS, fence, and $\forall u \in S$, comm. $u$ | |

**Table 5.1:** Summary of FIB ownership state transitions initiated by a barrier in thread $t$, coordination necessary to complete the transition, and whether the transition could raise a data-race exception. This table imitates Table 1 in [23] to facilitate comparison with OCTET.

[a]All three parts of this dissertation have food references.

[b]The term *conflict* here refers to conflicting metadata accesses rather than conflicting data accesses. A single-conflict transition may occur between a state Exclusive($u$), that arose due to a read barrier, and another state due to a second read barrier. The two corresponding data read accesses do not conflict, but updates in the access history by the read barriers do conflict.

updates only its own entry). The main access history record remains read-shared, but the individual entries of the auxiliary read map record are thread-private, but mutable.

Note also that, while an Exclusive($t$) state can be observed atomically, a Shared($S$) state cannot. The fact that the state is Shared is atomically observable, but the up-to-date set of sharing threads $S$ is not.

### 5.3.3 State Transition Overview

Ownership state transitions occur at every barrier, although in practice, most barriers cause self-transitions that do not change the ownership state of the access history in use. Table 5.1 summarizes the FIB state transitions that a given barrier may initiate and groups transitions both by how fast and common they are in practice and by the type of coordination they require. For each transition, the table also shows: the kind of barrier that initiates the transition; the starting and ending states of the transition; the coordination required for the transition; and whether the transition may detect a data race.

When a thread executes a barrier on an access history with a compatible ownership state, a self-transition is completed locally with no coordination with other threads. Self-transitions correspond exactly to those cases in FastTrack where data-race checks only reason about program order with earlier accesses by the current thread. Self-transitions never detect data races and hence never fail.

When a thread executes a barrier on an access history with an incompatible ownership state, a transition to a new state is attempted via coordination with other threads. State transitions correspond exactly to those cases in FastTrack where data-race checks require reasoning about synchronization with earlier accesses by other threads. State transitions may detect data races and fail. A state transition fails only if a data race occurs involving the access that initiated the transition. If two accesses race, one or both of their barriers will experience a transition failure and raise a data-race exception.

State transitions demand coordination with all threads that rely on guarantees of the current ownership state that would be invalid under the new state, to ensure that these threads are aware of the invalidation and that the transition checks against an up-to-date version of the access history to avoid missing data races. Coordination for state transitions requires either direct cross-thread communication or indirect coordination via a memory fence. While communication is very expensive compared to the cost of the single memory access it is deployed to analyze, transitions requiring communication are rare in practice. In exchange for the high cost of these rare transitions, FIB allows the common self-transitions to execute completely free of synchronization overhead.

Transitions are further grouped by the four types of coordination required:

- **Local:** When an access history is in state $\mathsf{Exclusive}(t)$ or $\mathsf{Shared}(S)$, where $t \in S$, a barrier by thread $t$ executes *locally*, with a self-transition. Self-transitions require no coordination and never detect data races. Local cases of the FIB barriers are described in §5.3.4.

- **Fence:** When an access history is in state $\mathsf{Shared}(S)$, a read barrier on this access history by thread $t \notin S$ requires a memory *fence* to coordinate a safe transition to state $\mathsf{Shared}(S \cup \{t\})$. Unlike all other (conflicting) non-local transitions, a successful transition from $\mathsf{Shared}(S)$ to $\mathsf{Shared}(S) \cup \{t\}$ preserves all permissions granted by $\mathsf{Shared}(S)$, so synchronous communication is not necessary for coordination. The fence serves to coordinate with a potential write barrier in some thread $u$ concurrently attempting to coordinate a transition to state $\mathsf{Exclusive}(u)$. We briefly discuss this transition in the read barrier along with local transitions in §5.3.4, since it is also in the fast path. We also discuss its interaction with the write barrier in §5.3.6.

- **Single-conflict:** When an access history is in state $\mathsf{Exclusive}(u)$, a write or read barrier on this access history by thread $t \neq u$ requires synchronous communication with the *single conflicting* thread $u$ to coordinate a safe transition to state $\mathsf{Exclusive}(t)$ or $\mathsf{Shared}(\{t, u\})$. We discuss the single-conflict coordination cases of write and read barriers and the supporting cross-thread communication mechanism in §5.3.5.

- **Multiple-conflict:** When an access history is in state $\mathsf{Shared}(S)$, a write barrier on this access history by any thread requires synchronization via atomic compare-and-swap, a memory fence, and communication with the *multiple conflicting* threads $u \in S$ to coordinate a safe transition to state $\mathsf{Exclusive}(t)$. We discuss the multiple-conflict coordination case of write barriers and the supporting cross-thread communication mechanism in §5.3.6.

Barriers can only execute locally when the access is data-race-free with respect to earlier accesses. Communication is a necessary (but not sufficient) condition to detect a data race. It is tempting to offer additional specializations in some cases where data races can be detected without expensive communication and without affecting accuracy, but in FIB's expected case—data-race-free execution—such "optimizations" only add expense to common cases, while never doing useful work.

### 5.3.4 Local Transitions

Local self-transitions align with the most common patterns of access in multithreaded programs: data-race-free accesses to data that is thread-private, at least temporarily, and data-race-free reads of data that is read-shared, at least temporarily. The write and read barriers, shown in Figures 5.5 and 5.6, respectively, test these cases first.

The write and read barriers both start by loading the `lastReads` field and checking if its value is a single last read by the current thread at `(A)` in Figures 5.5 and 5.6. This sequence serves identically as a lookup and check of ownership and a lookup and check for the first potential proof that the access is data-race-free.

#### 5.3.4.1 Exclusive Writes and Reads

If there is a single last read and it is by this thread, then: the access history is exclusive to this thread, so this check and any updates it does will execute atomically; and the access will be data-race-free with respect to all earlier accesses, by transitivity via program order with the last read and the invariant that all access so far happen before the single last read (§5.2.2). In this case, the barrier completes by updating the access history's last read—and, if writing, the last write—to this thread's current epoch.

#### 5.3.4.2 Shared Reads

Read barriers may also execute locally when the access history is Shared, *i.e.*, when there are multiple last reads represented by a read map. We break local shared barriers into three cases: `(B)`, `(C)`, and `(D)` in Figure 5.6. Examining case `(C)` first, if the ownership state is Shared($S$), where the current thread $t \in S$, then thread $t$ already has an entry[4] in the read map and has already done at least one data-race-free read that happens after the last write. The current access is therefore data-race-free with respect to earlier writes by transitivity via program order with the read recorded by thread $t$'s entry in the read map and the invariant that all other writes so far happen before all last reads recorded in the access history (§5.2.2). In this case, the barrier completes by updating thread $t$'s entry in the read map to the current epoch. Any thread attempting to write concurrently in Shared state will be responsible for communicating with thread $t$ (and all threads $u \in S$) to catch potential data races. Case `(B)` specializes case `(C)` to skip redundant updates of the current

[4]The lack of an entry is encoded as an entry of $0@t_0$, thus "has an entry" means "has an entry, $e \neq 0@t_0$."

```
write(t, history) {
  r = history.lastReads;

  // (A) EXCL(t) and DRF: last access was by t.
  if (isExclusive(r) && thread(r) == t)) {
    history.lastWrite = t.epoch;
    history.lastReads = t.epoch;
    return;
  }

  // (C) EXCL(u != t)
  if (isExclusive(r)) {
    request(r, t, history, WRITE);
    return;
  }
  // (D) SHARED
  writeShared(t, history, r);
}
```

**Figure 5.5:** FIB write barrier. `write(t, history)` takes the current thread `t` and access `history`.

epoch.

In case `(D)`, handled in `readSharedFirst(...)`, the ownership state is Shared($S$), where the current thread $t \notin S$: there are multiple last reads but none by this thread. This case requires a vector-clock check against the last write, since there is no earlier read by this thread since the last write that would establish transitive happens-before ordering. This case reverses the usual order of the data-race check and access history update and introduces a memory fence to avoid expensive communication without missing data races. We describe the interaction of this transition and potentially racing transitions to Exclusive($u$), where $u \neq t$, in §5.3.6. The use of a fence for threads $t \notin S$ in this case is analogous to OCTET's use of read-sharing counters to determine whether a read under read-shared ownership requires a fence [23].

```
read(t, history) {
  r = history.lastReads;

  // (A) EXCL(t) and DRF: last access was by t.
  if (isExclusive(r) && thread(r) == t.epoch)) {
    history.readWord = t.epoch;
    return;
  }

  if (isShared(r)) {
    // (B) SHARED({t, ...}) and DRF:
    // t has read in current epoch.
    if (r[t] == t.epoch) return;

    // (C) SHARED({t, ...}) and DRF:
    // t has read since last write.
    if (r[t] != NONE) {
      r[t] = t.epoch;
      return;
    }

    // (D) SHARED(S), but t not in S.
    // t has not read since last write.***
    readSharedFirst(t, history, r);
    return;
  }

  // (F) EXCL(u != t)
  request(r, t, history, READ);
}

readSharedFirst(t, history, r) {
  // Optimistic transition to SHARED({t} + S).
  r[t] = t.epoch;
  fence();

  // Check for racing write.
  w = history.lastWrite;
  if (w > t.vc[thread(w)]) {
    // Race.  Roll back to SHARED(S).
    r[t] = NONE;
    raise DRE;
  }
}
```

**Figure 5.6:** FIB read barrier. `read(t, history)` takes the current thread `t` and access `history`. `readSharedFirst(t, history, r)` also takes a read map `r`.

#### 5.3.4.3 Optimizations

To eliminate redundant access history updates, we specialize a few common cases of the local read and write barriers, following our version of FastTrack (§5.2). Figure 5.7 shows pseudo-code for these specializations to write and read barriers in `writeOpt(...)` and `readOpt(...)`, respectively.

**Write** The second specialization of the write barrier is simplest to explain and justify: it eliminates a redundant update of the last read when the last read is the current epoch but the last write differs. This check and the update of the last write still occur atomically, since the last read is an epoch belonging to this thread, giving it exclusive ownership.

The first specialization is from FastTrack: when executing a write barrier, if the executing thread has already written to this location during its current epoch (*i.e.*, the last write is the current epoch), then this barrier will be redundant with the barrier for the earlier write in the same epoch. Intuitively, it is impossible for any access in another thread to race with only one of these two writes, so any further check and update for this access is redundant. This specialization is an exception to the rule of checking the last read(s) to determine the history's ownership state. Nonetheless, it respects the ownership state of the access history: if the recorded last write is the current epoch, then the recorded last read is guaranteed to be the current epoch as well.

When a barrier sets a history's last write, it also sets the last read identically. An epoch is stored as a last write only after a successful data-race check on that epoch, so the equality of the current epoch and the last write guarantees that a write in the current epoch does not race with any earlier accesses. Clearly no thread has successfully written in a later epoch, otherwise the last write would hold a different epoch. Furthermore, no thread has stored a later last read since the last write. The current thread remains in the same epoch as the last write, so it cannot have stored a newer epoch as the last read. Any other thread $u \neq t$ must synchronize with thread $t$ after thread $t$'s current epoch for any access in $u$ to pass the data-race check with the last write and record a new last read, but such synchronization is impossible. Any outgoing synchronization from thread $t$ would increment thread $t$'s current

```
writeOpt(t, history) {
  // EXCL(t) and DRF:
  // last write was by t in current epoch.
  // Also implies history.lastReads == t.epoch.
  if (history.lastWrite == t.epoch) return;

  // EXCL(t) and DRF:
  // last access was by t in current epoch.
  if (history.lastReads == t.epoch) {
    history.lastWrite = t.epoch;
    return;
  }

  // General cases.
  write(t, history);
}

readOpt(t, history) {
  // EXCL(t) and DRF:
  // last access was by t in current epoch.
  if (history.lastReads == t.epoch) return;

  // General cases.
  read(t, history);
}
```

**Figure 5.7:** Specializations of the Exclusive cases of local FIB barriers. `writeOpt(t, history)` and `readOpt(t, history)` both take the current thread `t` and access `history`.

epoch such that it would be different from the last write, which is a contradiction. Thus it is guaranteed that the last read has not changed since it was set to this thread's current epoch along with the last write, so the ownership state is Exclusive($t$).

**Read** The read barrier specialization (from FastTrack) skips a redundant update of the last read when it is identical to the current epoch, yielding a common fast path with no metadata updates.

### 5.3.5 Single-Conflict Transitions

To support communication for single-conflict coordination, each thread maintains a queue of incoming requests and a field to store an incoming response. When a barrier in thread $t$ encounters an access history in state $\mathsf{Exclusive}(u)$, where $u \neq t$, thread $t$ requests that thread $u$ perform a data-race check, update, and state transition on its behalf. We refer to this requested work as the *check-and-transfer* sequence. If thread $u$ is running, the request is enqueued with thread $u$ and thread $t$ waits for thread $u$ to respond. If thread $u$ is blocked, thread $t$ self-serves its own request, just as in OCTET [23].

When a thread reaches a *yield point* in execution, it processes all the requests in its queue, and responds to each with the data-race check result. Yield points must occur within a bounded interval to ensure forward progress. They are typically inserted in application code at calls, returns, loop back-edges, blocking operations, etc. Managed language implementations typically already have such yield points for run-time services such as garbage collection and on-stack replacement [50].

#### 5.3.5.1 Request Dispatch and Response Handling

A thread $t$ can successfully process a request only if the requested access history is in state $\mathsf{Exclusive}(t)$, for the same reason that the requesting thread was not allowed to perform a single-conflict state transition without coordination in the first place. Requesting threads atomically look up—and enqueue a request with—an access history's current owner thread, to ensure that *check-and-transfer* requests are sent only to the correct owner thread even if other state transitions occur concurrently. Once a request has arrived at the current owner thread, it will eventually be processed. Pseudo-code for sending a check-and-transfer request and handling the response is shown in Figure 5.8.

To send a request to thread $u$, thread $t$ first locks thread $t$'s queue and re-examines the ownership (by reloading the last read(s)) of the access history it will request, to guard against protocol races.

In the common case, the access history is still owned by thread $u$. If thread $u$ is running, thread $t$ enqueues a request in thread $u$'s queue, unlocks $u$'s queue, and awaits a response

```
request(r, t, history, kind) {
  do {
    u = thread(r);
    lock u.queue;
    r = history.lastReads;
    if (isExclusive(r) && thread(r) == u) {
      if (u.blocking) { // self-service
        result = checkAndTransfer(t, history, kind);
        unlock u.queue;
      } else { // send request
        enqueue(u.queue, (t, history, kind));
        unlock u.queue;
        result = awaitResponse(t);
      }
      if (result == OK) return;
      else raise DRE;
    } else {
      unlock u.queue;
    }
  } while (isExclusive(r));

  // Concurrently changed to SHARED.
  if (kind == READ) {
    readSharedFirst(t, history, r);
  } else { // kind == WRITE
    raise DRE;
  }
}
```

**Figure 5.8:** FIB request dispatch and response handling in communicating barriers. `request(r, t, history, kind)` takes the expected value `r` of the access history's last-reads field, the requesting thread `t`, the access history `history`, and the `kind` of access (`READ` or `WRITE`).

from thread $u$. If thread $u$ is blocked, thread $t$ self-serves its own request without enqueueing anything and unlocks $u$'s queue. The same check-and-transfer sequence is executed in both cases. We discuss this data-race checking logic (`checkAndTransfer(...)` in Figure 5.9) shortly.

In the rare case, a concurrent barrier on the same access history may cause the ownership state to change by the time thread $t$ has locked thread $u$'s queue, causing the attempted atomic lookup-and-enqueue to fail. In this case, thread $t$ unlocks thread $u$'s queue without enqueueing a (stale) request. If thread $t$ is writing, a concurrent ownership change clearly indicates a data race. If thread $t$ is reading, a concurrent ownership change not indicate a data race. Additional consideration of ownership and possible new communication attempts will be required to resolve the data-race check.

- If the access history has become exclusive to a thread other than thread $u$ (note that it cannot have become exclusive to $t$), then thread $t$ restarts the *check-and-transfer* process with this new thread (repeating the loop in Figure 5.8) to check whether it read (safe) or wrote (race).

- If the access history has become shared and the last read(s) holds a read map, then no communication is needed and thread $t$ continues (falling out of the loop in Figure 5.8) by performing a check and update against this read map, following the previously-discussed `readSharedFirst(...)`, from Figure 5.6.

Finally, although it is not shown explicitly in the pseudo-code (hidden in `awaitResponse`), a thread must respond to incoming requests (or allow other threads to self-serve requests) while waiting for a response to its own request to avoid deadlock via a cycle of requests.

The atomic lookup-and-enqueue as described here and in Figure 5.8 may experience starvation, as it essentially uses safe double-checked locking in a loop to ensure atomicity. Starvation has never occurred in practice and failed atomic lookup-and-enqueue operations are rare, as discussed in §5.8.2. Approaches with non-atomic lookup-and-enqueue operations would have to handle stale requests sent to threads that no longer own the requested access histories. Recognizing the need to retry becomes subject to remote communication latency rather than local lookups, making a non-atomic lookup-and-enqueue more susceptible to starvation.

#### 5.3.5.2 Check and Transfer

Regardless of whether a request is served by a remote thread or self-served when the remote thread is blocked, the same check-and-transfer logic is applied. Figure 5.9 shows pseudo-code. These checks are most of the cases in FastTrack that compare an accessing thread's vector clock with an access history that has a single last read. (Checks against multiple last reads are covered with multiple-conflict coordination, in §5.3.6.) These checks are used in contexts satisfying the following invariants:

- Earlier in the same barrier, the access history was observed to have ownership state $\mathsf{Exclusive}(u)$.
- No thread will change this access history concurrently with this check. Either this check is run while the current owner thread, $u$, is blocked or this check is run by the initially observed owner thread $u$ itself and this access history is a private copy.
- If the access history has been changed since the earlier observation of its ownership state, $\mathsf{Exclusive}(u)$, then the change was performed by thread $u$ in response to a request concurrent with this one, and this access history is a private copy, and thread $u$ is running this check.

**Write** A write by thread $t$ in this context is data-race-free with respect to all previous accesses if there is a single last read by some thread $u$ (*i.e.*, the access history has ownership state $\mathsf{Exclusive}(u)$) and that single last read is no newer than thread $u$'s entry in thread $t$'s vector clock. Otherwise the write creates a data race. If there is no data race, the last write and last read are updated to thread $t$'s current epoch, which also causes the access history's ownership state to become $\mathsf{Exclusive}(t)$.

The general version of FastTrack also determines a write to be data-race-free if it follows multiple concurrent last reads and happens after all of those last reads. However, in the

```
checkAndTransfer(t, h, k) {
  r = h.lastReads;
  if (k == WRITE) {
    if (isExclusive(r) && r <= t.vc[thread(r)]) {
      h.lastWrite = t.epoch;
      h.lastReads = t.epoch;
      return OK;
    } else {
      return RACE;
    }
  } else { // k == READ
    if (isExclusive(r)) {
      if (r <= t.vc[thread(r)]) {
        h.lastReads = t.epoch;
        return OK;
      } else if (h.lastWrite <= t.vc[thread(h.lastWrite)]) {
        h.lastReads = { (thread(r), r), (t, t.epoch) };
        return OK;
      } else {
        return RACE;
      }
    } else { // isShared(r)
      if (h.lastWrite <= t.vc[thread(h.lastWrite)]) {
        r[t] = t.epoch;
        return OK;
      } else {
        return RACE;
      }
    }
  }
}
```

**Figure 5.9:** FIB check-and-transfer handling. `checkAndTransfer(t, h, k)` takes the requesting thread `t`, access history `h`, and kind of access `k`.

context where this check is used, we observed earlier in the barrier that the access history had a single last read. Thus if it now has multiple last reads, one of those reads was clearly concurrent with this write. Comparing each last read against thread $t$'s vector clock will definitely reveal at least one data race, so we omit the comparison since its result is known and immediately report a data race. In fact, *any* change to the access history since this barrier's initial observation indicates a data race. However, a lack of change does not demonstrate data-race freedom, so there are no optimization opportunities for data-race-free cases.

**Read** A change in the number of last reads in this access history (from Exclusive to Shared) does not indicate a data race, so all FastTrack read cases that compare vector clocks with access histories also appear here.

If the access history is exclusive to some thread, and the single last read is in an epoch of thread $u'$ that happens before thread $u'$'s entry in thread $t$'s vector clock, then there is no data race and the single last read is replaced by the current epoch of thread $t$. The new ownership state is $\mathsf{Exclusive}(t)$. (Furthermore, $u' = u$, by the invariants in §5.2.2.)

If the access history is exclusive to thread $u'$, and the single last read does not happen before thread $u'$'s entry in thread $t$'s vector clock, and the last write, by thread $u''$, happens before thread $u''$'s entry in thread $t$'s vector clock, then there is no data race and the single last read is replaced by a read map mapping $u'$ to the current last read and $t$ to thread $t$'s current epoch. The new ownership state is $\mathsf{Shared}(\{u', t\})$.

If the access history is in a Shared state and the last write, by thread $u'$, happens before thread $u'$'s entry in thread $t$'s vector clock, then there is no data race and a mapping from $t$ to thread $t$'s current epoch is added to the read map. Furthermore, by the invariants, thread $t$ must not have had an entry in this read map before it was added here, since this barrier earlier observed the access history to be $\mathsf{Exclusive}(u)$ and the necessary changes to arrive at a Shared state all come from requests concurrent to the one served here.

In all other cases, this read creates a data race with the last write.

```
yield(t, blocking) {
  lock t.queue;
  for history s.t. (u, history, k) in t.queue {
    // Accumulate responses and history updates in private copy.
    responses = {};
    hc = copy(history);
    for (u, k) s.t. (u, history, k) in t.queue {
      responses += (u, checkAndTransfer(u, hc, k));
    }
    // Publish history updates.
    history.lastWrite = hc.lastWrite;
    history.lastReads = hc.lastReads;
    // Respond.
    for (u, resp) in responses { respond(u, resp); }
  }
  for u in t.acks {
    respond(u, ACK);
  }
  if (blocking) t.blocking = true;
  unlock t.queue = {};
}
continueAfterBlocking() {
  lock t.queue;
  t.blocking = false;
  unlock t.queue;
}
```

**Figure 5.10:** FIB request processing at yield points. `yield(t, blocking)` is called at yield points of thread `t` with an indication of whether the thread is will be `blocking`. `continueAfterBlocking()` is called after returning from blocking to start accepting requests again.

#### 5.3.5.3 Queue Processing and Response

Figure 5.10 shows pseudo-code for request processing at yield points. A thread locks its queue during request processing to prevent new requests while processing the existing queue. An alternative approach is to allow concurrent requests, processing the queue until it becomes empty or through some arbitrary maximum number of requests. We choose the former approach to simplify the implementation of atomic lookup-and-enqueue for requesting threads.

Rather than processing and responding to each request individually, a responding thread groups requests by access history. For each group, the responding thread processes the group of requests using a private copy of the access history. After processing all requests in the group, it publishes the final resulting access history and sends responses to all requests in the group. This avoids turning queued requests stale (*i.e.*, enqueued with a thread that cannot process the request safely) during processing of other requests earlier in the queue, along with the concomitant issues of forwarding, latency, and starvation.

Consider an access history with state $\mathsf{Exclusive}(u)$ when threads $t$ and $t'$ both enqueue requests to read. Suppose the request from thread $t$ is processed first and causes the transition $\mathsf{Exclusive}(u) \rightarrow \mathsf{Exclusive}(t)$. If this update is published to the access history immediately, the request from thread $t'$ is stale by the time thread $u$ reaches it, as thread $u$ no longer owns the access history. If thread $u$ also sends a response immediately, thread $t$ could execute further barriers on this access history under the valid assumption that it holds exclusive ownership. Even if thread $u$ defers its response to $t$ until the end of queue processing, but still publishes access history updates immediately, other threads could observe the $\mathsf{Exclusive}(t)$ ownership state and self-serve requests for this access history with thread $t$ while thread $t$ is blocked waiting for a response from thread $u$.

By using a private copy of the access history and deferring its publication and the delivery of responses to the end of queue processing for that access history, thread $u$ can make multiple state transitions even into and through states where it does not have permission to do checks or updates under the normal protocol. All other threads still see the global copy of the access history as $\mathsf{Exclusive}(u)$ until it is updated with the private copy, blocking until the new copy is published or thread $u$'s queue is unlocked after processing. This prevents stale requests during queue processing.

In fact, all sequences of transitions possible from a set of requests for a single access history at a single yield point are described by three patterns. In the following, *last write* and *last read* refer to the access history at the beginning of request processing, before any updates have been made. Request processing is being done by thread $u$ and the access history is initially in state $\mathsf{Exclusive}(u)$.

- $\mathsf{Exclusive}(u) \rightarrow \mathsf{Exclusive}(t)$

A single write request or a single read request that happens after the last read.

- $\mathsf{Exclusive}(u) \to \mathsf{Shared}(S_0) \to^* \mathsf{Shared}(S_n)$
  where $\exists t, S_0 = \{u, t\}$ and $n \geq 1$ and $\forall i \in 1..n, \exists t' \notin S_{i-1}, S_i = S_{i-1} \cup \{t'\}$.

  One or more concurrent read requests that happen after the last write and at least the first of which is concurrent with the last read.

- $\mathsf{Exclusive}(u) \to \mathsf{Exclusive}(t) \to \mathsf{Shared}(S_0) \to^* \mathsf{Shared}(S_n)$
  where $\exists t', S_0 = \{t, t'\}$ and $n \geq 1$ and $\forall i \in 1..n, \exists t'' \notin S_{i-1}, S_i = S_{i-1} \cup \{t''\}$.

  One or more concurrent read requests that happen after the last write and at least the first of which happens after the last read.

Our actual implementation exploits these patterns and the fact that a transition $\mathsf{Shared}(S) \to \mathsf{Shared}(S \cup \{t\})$ is accomplished by thread-private mutation in the read map with no mutation to the main access history. We slightly decrease response latency by publishing the private access history and enabling immediate responses whenever an access history reaches a Shared state, rather than waiting to the end of request processing. Any remaining requests for that access history are served against the same read map.

### 5.3.6 Multiple-Conflict Transitions

Multiple-conflict transitions occur in write barriers following read-sharing. This case is rare in practice and requires the most expensive form of coordination: communication with multiple threads. Figure 5.11 shows pseudo-code for this case of the write barrier (`writeShared(...)`) plus supporting communication infrastructure (`ack(...)`). This section also considers interaction with read barriers in Shared states (see Figure 5.6).

Read and write barriers for Shared ownership are carefully co-designed to minimize overhead in the common read-shared case at the cost of overhead in the rare write-after-read-shared case. Most instances of the read barrier in Shared state will never examine the last write, relying on the knowledge that (1) the access history is in Shared state and (2) the presence of an entry for the current thread in the corresponding read map demonstrates

```
writeShared(t, history, r) {
  // Then take tentative ownership.
  if (!CAS(&history.lastReads, r, t.epoch)) raise DRE;

  // Tentatively install new last write.
  w = history.lastWrite;
  history.lastWrite = t.epoch;
  fence();

  // Ack and check existing last reads.
  for (u, _) in t.vc {
    if (r[u] != NONE) {
      ack(u, t);
      // If data race, roll back -> SHARED(r).
      if (r[u] > t.vc[u]) {
        history.lastWrite = w;
        history.lastReads = r;
        raise DRE;
      }
    }
  }
}
ack(u, t) {
  lock u.queue;
  if (u.blocking) {
    unlock u.queue;
    return;
  } else {
    enqueue(u.acks, t);
    unlock u.queue;
    awaitResponse(t);
  }
}
```

**Figure 5.11:** FIB write barrier case for Shared state. `writeShared(t, history, r)` takes the current thread `t`, the current access history `history`, and the value `r` loaded from the last-reads field earlier in the write barrier. `ack(u, t, h)` take the thread `u` from which to request an ack and the current thread `t`.

that an earlier access by this thread happened after the last write, so by transitivity via program order, the current read also happens after the last write. This reasoning relies on the assumption that a transition out of Shared state to an Exclusive state, with an accompanying update of the last write, cannot concurrently invalidate the above reasoning without also detecting a data race. To support this assumption, the write barrier's case for Shared state is structured as follows.

First, a write barrier in thread $t$ finding a Shared access history preemptively CASes its current epoch into the access history's last read field to make a tentative state transition to Exclusive($t$). This ensures that read barriers sufficiently far in the future will see the new Exclusive($t$) state and commence communication with this thread which will detect any data races. Additionally, a failure of this CAS indicates that at least one other concurrent write barrier is racing to make a transition out of Shared state, detecting a write-write race. The CAS guarantees that only one Shared $\rightarrow$ Exclusive transition is underway at any time. However, this preemptive ownership transition may temporarily break an invariant of the access history: namely, that all other access in the execution so far happen before the recorded last read. The write barrier's atomic ownership transition may also occur between a concurrent read barrier's observation of Shared state and the read barrier's update of a read map entry ((C) in Figure 5.6), causing the read barrier to miss a data race. Rather than adding communication to common read barrier cases, the write barrier compensates to detect these races in a way that exploits the careful guarantee that *one or both* (not necessarily *exactly* one) of a pair of racing accesses will raise an exception.

After taking tentative ownership of the access history, the write barrier updates the last write to the current epoch and issues a memory fence to make the new tentative last write visible before it continues. The write barrier keeps local copies of the old last write and old read map for coordination purposes and for potential rollback if this barrier detects a data race.[5]

Next, the write barrier communicates with all threads that currently have an entry in the read map that it CASed out of the access history earlier. For the purposes of this case

[5]Rollback is clearly unnecessary under halt-on-first-race semantics. If only the offending thread is to receive an exception and others may continue, then rollback is needed.

of the write barrier, we use a separate communication mechanism which shares some of the infrastructure described earlier, but simply provides an acknowledgment or *ack* to the requesting thread that the responding thread has reached a yield point, guaranteeing that:

- The responding thread will see any access history updates the requesting thread performed prior to its request.

- The requesting thread will see any access history updates the responding thread performed before its response.

Here, in state $\mathsf{Shared}(S)$, the write barrier in thread $t$ requests acks from all other threads $u \in S$, *i.e.*, all other threads that have recorded a last read in the read map. Receipt of this ack guarantees that:

- The responding thread will see the new $\mathsf{Exclusive}(t)$ state in any future barriers, thus avoiding missed data races.

- The writing thread will see the most recent entry from the responding thread in the read map from the preceding Shared state.

While waiting for an ack from another thread, this thread may need to take yield points and respond to other threads' requests to avoid deadlocked communication requests. In doing so, it may receive requests for this access history. The very existence of such a request clearly indicates a data race, but processing the request as usual will detect the race, so we do not specialize this unlikely case.

After receiving a thread's ack, the write barrier does the conventional happens-before check for that thread, comparing its entry in the read map to the corresponding entry in this thread's vector clock.[6] If a race is detected, then the write barrier rolls back the access history to its earlier state, replacing the last write and last read map. Note that the racing read barrier may also detect this race, which is allowed by our accuracy guarantee.

[6]The simplified pseudo-code for requesting and acting on acks is over-constrained. The write barrier may issue all ack requests asynchronously, and then process the responses in any order.

If no data races were detected then the barrier completes, as the current epoch has already been installed as the last write and last read of this access history.

#### 5.3.6.1 Interaction with Fence Transitions

Even if the write barrier succeeds, other threads without entries in the last-reads map may concurrently perform communication-free fence transitions in read barriers. Under the conventional order of data-race check followed by access history update, a racing read barrier could check against the old Shared state *before* the write barrier makes the transition to Exclusive$(t)$ and add its entry to the read map *after* the write barrier has finished its communication. Thus neither barrier catches the data race! Even if the write barrier requests acks from all live threads instead of just those found in the read map at the time, this missed update could still occur.

This case motivates the co-design of the read barrier case in which a thread $u \notin S$ reads in Shared$(S)$ state, *i.e.*, reads in Shared state, but for the first time since the last write, when it does not have a read map entry. This case is covered at `(D)` and in `readSharedFirst(...)` in Figure 5.6 and in §5.3.4. To review, this read barrier case first tentatively stores its current epoch in the read map and issues a memory fence to ensure this entry is visible. Then, it does a happens-before check against the last write in the access history. If this check succeeds, the barrier is complete. If it fails, the barrier rolls back its tentative update. With both the shared case of the write barrier and this first-time shared case of the read barrier updating the access history before doing data-race checks, it is guaranteed that when these two barriers race, one or both of the barriers will detect that race.

- If the write barrier's last-read CAS and last-write update do not become visible to the read barrier before the read barrier checks the ownership state and the last write, then the read barrier's read map update must be visible to the write barrier by the time the write barrier checks that entry in the read map when doing communication and happens-before checks with threads that do have entries.

- If the read barrier's read map update does not become visible to the write barrier until after the write barrier has checked that entry in the read map, then the write barrier's last-write update must be visible to the read barrier by the time it checks against the last write.

The tentative access history updates in these cases of the write and read barriers do not cause missed data races. Data-race checks exploit the transitivity of the happens-before relation to reduce performance overhead while avoiding missed races. To do this, they rely on the invariant that, at a given point in an execution, all accesses to a data location in the execution so far happen before at least one of the reads presently recorded in its access history. Tentative access history updates may temporarily break this invariant by updating the access history before checking that the history it overwrites happens before the new access it records. Thus other data-race checks that see the tentatively updated access histories could falsely assume that all earlier accesses happen before this current access history and miss data races with earlier accesses. However, this transitivity assumption is only applied when there is happens-before order with the accesses currently recorded in the access history, which is never the case for tentative access history updates. The transitivity invariant is restored by the end of a barrier that makes tentative access history. By the time another thread can observe that the tentative last read or last write happens before its current access (*i.e.*, there has been synchronization *after* the tentatively updating barrier), the transitivity invariant is restored, so this other thread cannot miss a true data race.

Tentative access history updates do not cause false data races, since tentative last reads or last writes represent real accesses. Thus the co-design of these two cases of the write and read barriers maintains soundness and completeness, with no missed or false data races, while allowing all read-shared barriers to execute without communication and, except for a single memory fence in some cases, without synchronization.

#### 5.3.6.2 Alternatives

The use of fence transitions in read barriers allows the multiple-conflict case of the write barrier to do coordination by communicating with only those threads that are members of the current Shared state. An alternative is to communicate with all live threads in the system and omit the fence transitions in read barriers. For large thread counts, our approach may have significantly better performance in this case of the write barrier, especially if sharing is often narrow (*i.e.*, few threads have shared) relative to the number of live threads in the system. However, this benefit is exercised only if the multiple-conflict case of the write barrier is not excessively rare. From our profiling results (see Table 5.2), it appears that this case is quite rare in practice.

### 5.3.7 Progress Guarantee

FIB guarantees that, if at least one thread is attempting to complete a FIB barrier, at least one thread will eventually complete its FIB barrier or raise a data-race exception.

FIB may cause *starvation*: there exist fair schedules of data-race-free programs under which one thread is unable to complete its FIB barrier, while other threads repeatedly complete FIB barriers. For example, starvation of thread $t_3$ can occur in a data-race-free program execution under the following conditions: an access history is initially $\mathsf{Exclusive}(t_1)$, and "should" eventually become $\mathsf{Shared}(\{t_1, t_2, t_3\})$, but instead it makes transitions back and forth between $\mathsf{Exclusive}(t_1)$ and $\mathsf{Exclusive}(t_2)$ as $t_1$ and $t_2$ complete alternating reads ordered by synchronization while $t_3$ never successfully completes an atomic lookup-and-enqueue to request concurrent access and force the transition to $\mathsf{Shared}(\{t_1, t_2, t_3\})$. There exist fair schedules that allow $t_3$ to complete its barrier, but there also exist fair schedules that do not.

FIB never causes starvation if all attempts to atomically lookup-and-enqueue requests eventually succeed. Regardless, the Java language specification makes no fairness guarantee [76], so the possibility of starved lookup-and-enqueue attempts should be indistinguishable from lack of fairness. In practice Java programs may falsely assume fairness, but in practice, atomic lookup-and-enqueue starvation has never occured.

FIB never introduces *deadlock* or *livelock* where it could not occur without FIB. It is always the case that at least one thread can complete its FIB barrier and access, assuming fair scheduling and atomic CAS operations that do not fail spuriously. If a barrier depends on a response to a request to another thread, that response will eventually arrive. Either the responding thread is running and will eventually reach a yield point where the request will be served, or the responding thread is blocked, in which case the request can be self-served by the requesting thread. The possibility for starvation of a single thread exists when trying to enqueue or self-serve a request, but depends on progress by other threads. If other threads are not making progress, starvation is impossible, and the requesting thread is guaranteed to make progress.

## 5.4 Extensions

We have implemented two extensions to the FIB protocol described in §5.3: dynamic thread-escape analysis to filter data-race checks and lazy initialization of access history ownership states.

### 5.4.1 Dynamic Thread-Escape Analysis

Conservative dynamic thread-escape analysis can be used as a pre-filter to skip data-race detector analysis barriers on data that are reachable by only one thread. This is likely to offer stronger benefits to pessimistically synchronized data-race detectors than to FIB, but may still help FIB's performance, since some updates to access histories are necessary, regardless of what synchronization is needed to protect them. When an access occurs in an epoch newer than the last read (or write if writing), the access history must be updated in case the data is eventually used by another thread, when this access history will be necessary for data-race detection. Conservative dynamic thread-escape analysis (§5.7.4) can elide even these updates for data that remains reachable from only one thread.

#### 5.4.1.1 Filtering Access Barriers

To use dynamic thread-escape analysis as a pre-filter for data-race detection barriers, the conservative analysis described in §5.7.4 is run independently of the data-race detector. Each data-race detector barrier starts by checking the escape status of the object holding the current access history's data location. If the object is Escaped, then the normal FIB barrier is executed. If the object is Private, then the fact that this thread can access it means that it is private to this thread and is not yet reachable by any other thread, so the FIB barrier is skipped.

**Completeness** Using the thread-escape pre-filter clearly does not introduce false data races. In general, by checking fewer accesses for data races it is only possible to miss true data races, not to introduce false data races.

**Soundness** Using the thread-escape pre-filter also does not cause FIB to miss true data races, under the first-race guarantee (§2.2.3). With filtering, accesses to location before it escapes are not recorded in its access history. After escape, accesses are checked and recorded.

Before escape, all accesses are by one thread, so there are clearly no data races.

The analysis will not detect a true data race between a post-escape access (outside the initial owner thread) and a pre-escape access, but if such a data race is missed then it is preceded by an earlier detected data race involving the access that published the reference to the escaping object. Figure 5.12 illustrates this scenario. The grayed-out data-race check in thread $t_2$ would fail to report a data race, but it is never reached because thread $t_2$'s earlier access, to acquire the reference $o$, also races with thread $t_1$ access which caused $o$ to escape.

Suppose object $o$ is initially private to thread $t_1$ and $t_1$ writes to field $o.\mathsf{f}$ before $o$ escapes. If thread $t_2$ can access a field $o.\mathsf{f}$ of an object that was previously accessed by thread $t_1$ while $o$ was private to thread $t_1$, then there must be an earlier access by thread $t_2$ which reads the reference to $o$ published by thread $t_1$ at $o$'s escape.

If thread $t_2$'s access of $o.\mathsf{f}$ races with thread $t_2$'s pre-escape access to $o.\mathsf{f}$, then it must be the case that there has been no synchronization from thread $t_1$ to thread $t_2$ between thread $t_1$ and thread $t_2$'s accesses to $o.\mathsf{f}$.

By program order, thread $t_1$'s pre-escape accesses to $o.\mathsf{f}$ happen before its escaping publication of a reference to $o$, which, since $o$ was not reachable by $t_2$ before this publication, causally precedes thread $t_2$'s acquisition of this reference, which, by program order, happens before thread $t_2$'s access to $o.\mathsf{f}$. Therefore, the accesses passing reference $o$ from thread $t_1$ to thread $t_2$ happen between the racing accesses to $o.\mathsf{f}$, so if there is no synchronization between the latter, there is no synchronization between the former, and the passing of the reference must also race. This race will be detected before the race on $o.\mathsf{f}$ can occur, satisfying the first-race guarantee regardless of whether the race on $o.\mathsf{f}$ is detected. With

| **Thread $t_1$ (in epoch $c_1@t_1$)** | **Thread $t_2$ (in epoch $c_2@t_2$)** |
|---|---|
| let $o$ = new . . . | |
| | |
| FIB: Is $o$ escaped? No, OK. | |
| set $o$.f = 1 | |
| | |
| ESC: Is $g$ escaped? Yes. | |
| ESC: Is $o$ escaped? No. | |
| ESC: Mark $o$ escaped. | |
| FIB: Is $g$ escaped? No. | |
| FIB: . . . data-race check . . . OK. | |
| FIB: Store $c_1@t_1$ in $W_{g.\mathsf{obj}}$. | |
| FIB: Store $c_1@t_1$ in $R_{g.\mathsf{obj}}$. | |
| set $g$.obj = $o$ | . . . |
| . . . | FIB: Is $g$ escaped? Yes. |
| | FIB: Load $c_1@t_1$ from $R_{g.\mathsf{obj}}$ |
| | FIB: . . . coordination . . . |
| | FIB: $c_1@t_1 \preceq C_{t_2}$? No. **Race!** |
| | FIB: . . . |
| | let $o$ = $g$.obj |
| | |
| | FIB: Is $o_2$ escaped? Yes. |
| | FIB: . . . data-race check . . . **OK.** × |
| | FIB: Store $c_2@t_2$ in $R_{o.\mathsf{f}}$. |
| | let $n$ = $o$.f |

**Figure 5.12:** A missed data race on $o$.f due to escape-filtering is preceded by a detected data race on $o$'s escape through $g$.obj. Time flows down. $o$ is used as a local variable in both threads, but refers to the same object. $g$ is a global variable. The missed data race follows detection of the first race on $g$.obj and is never reached if data races are exceptions.

data-race exceptions, the second race (and all of the gray section of thread $t_2$'s execution in Figure 5.12) will never be reached nor missed.

Thus eliding data-race checks on private data can result in missed true data races only if they follow at least one reported true data race. This behavior meets the first-race guarantee.

#### 5.4.1.2 Filtering Synchronization Instrumentation

It is also possible to elide a vector-clock race detector's instrumentation of synchronization operations (*e.g.*, lock acquire and release) when they operate on synchronization state in thread-private objects without missing data races or introducing false data races. Clearly if we ignore synchronization, we cannot miss data races, since we have only reduced the perceived happens-before ordering of accesses. The argument for the lack of false data races here is similar to that against missed data races when using escape status to filter data-race checks on accesses.

Incoming (acquire) synchronization on private synchronization state (lock) never grows the happens-before order of a program, because the synchronization order due to these operations is subsumed by program order in the single thread that can use the object. Only the last outgoing (release) synchronization on a given piece of private synchronization state can possibly link up with a later incoming synchronization operation in a different thread. All early outgoing synchronization on this synchronization is necessarily private and happens before the final instance. Thus with locks, for example, we need only consider the last release of a lock before that lock escapes.

Ignoring the last pre-escape release of lock $l$ by thread $t_1$ could cause a data-race detector to miss transitive happens-before order between accesses of $t_1$ happening before this release operation and accesses of some thread $t_2$ happening after a later acquire of lock $l$, resulting in false data races reported in thread $t_2$. However, false data races can only be reported here if this pair of release and acquire formed the *only* happens-before order between these accesses in threads $t_1$ and $t_2$. Following the argument for eliding data-race checks, if this is indeed the case, then there exists an earlier true data race between thread $t_1$'s escaping publication of a reference to lock $l$ and thread $t_2$'s later observation of this reference. The synchronization on lock $l$ cannot be responsible for ordering these accesses, since thread $t_2$

needs the reference to lock $l$ before it can acquire lock $l$.

Thus eliding instrumentation of synchronization operations on private synchronization state can only result in reported false data races if they follow at least one reported true data race. This behavior meets the first-race guarantee.

### 5.4.2 Ownership State Initialization

The Fib protocol assumes that every access history is initialized with ownership state $\mathsf{Exclusive}(t)$ when allocated by thread $t$. In practice, we do not use allocation-time initialization, implementing one of the two following deferred initialization policies instead. Every barrier using a given access history must follow the same policy. In practice we use one policy exclusively.

#### 5.4.2.1 CAS for Initial Ownership

The simplest ownership state initialization method is lazy. Access histories are allocated in a third ownership state, $\mathsf{None}$, implicitly encoded by the origin epoch $0@t_0$ in the last read. When a thread finds an access history in state $\mathsf{None}$, the access history has not been used before. A thread $t$ must make a transition to $\mathsf{Exclusive}(t)$ in order to proceed with a barrier. To make this transition, it uses an atomic compare-and-swap to set the last read to its current epoch. If this CAS fails in a write barrier, there is definitely a data race. If it fails in a read barrier, it may or may not be a data race, depending on the type of concurrent barrier that caused it to fail, so the read barrier falls back to the normal Fib protocol, looking up the new ownership state and coordinating with the owner thread.

CAS for initial ownership may be used with or without thread-escape filtering.

#### 5.4.2.2 Indirect Initial Ownership via Thread-Escape Analysis

When conservative dynamic thread-escape analysis is used a pre-filter to Fib's barriers, it can support CAS-free deferred initialization of access history ownership state. Since the Fib barrier is never run on an access history until after the containing object has escaped, the access history's ownership state need not be initialized until escape. If an object never escapes, its access histories never need their ownership states initialized.

We replace the Escaped flag with Escaped($t$), where thread $t$ is the thread to which the object was originally private. The escape check remains as a pre-filter to a slightly modified FIB barrier. We call this an *eager* escape check. If the object containing the access history is Escaped($t$), then the FIB barrier executes. When deriving the ownership state from the access history's last reads, a last read of $0@t_0$ means the access history is Exclusive($t$), where the object containing the access history is Escaped($t$), as determined earlier in the escape check. Barriers on access histories for static fields (global variables) still require the CAS-for-ownership initialization, since they are never private, and thus have no escaping thread to lend them an initial ownership state.

The escape check can be reordered into the FIB barrier instead of used as a pre-filter. We call this a *lazy* escape check. In this case, the FIB barrier executes on every access. When looking up the last read to determine ownership, if the last read is a non-zero epoch or a read map, the FIB barrier executes normally without an explicit escape check. If the last read is $0@t_0$, then the containing object's escape status is checked. If the object is Private, then the rest of the FIB barrier is skipped. If the object is Escaped($t$) then we derive the ownership state Exclusive($t$) and execute the FIB barrier accordingly.

Eager escape checks are expected to have slightly better performance when most accesses are to objects that are not escaped, since these require looking up just the escape status. Lazy escape checks are expected to have slightly better performance when most accesses are to objects that are escaped, since these can omit looking up the escape status on non-empty access histories. The decision between eager and lazy escape checks can be made individually for each (static) barrier. In practice, we have used only eager escape checks.

## 5.5 Implementation

We implemented FIB and other versions of the FastTrack algorithm [52] in the x86 version of the Jikes RVM Java virtual machine [8], version 3.1.3 as extended by Bond, *et al.*, in their implementation of OCTET [23]. Of the extensions in the OCTET code base, we use only the facilities for inserting barriers. While OCTET and FIB execute similar protocols, their details differ enough that they do not currently share implementations of communication, etc. (See §5.7.3.)

### 5.5.1 Common Metadata and Instrumentation

All of our FastTrack implementations share the same access history metadata, synchronization metadata, and synchronization tracking instrumentation. They differ only in the code for access barriers and yield points.

#### 5.5.1.1 Metadata

An access history is encoded as two adjacent words in memory, the *last-read word* and the *last-write word*. Access histories for object fields are laid out inline in the same objects as the fields they shadow. Access histories for static fields are laid out in the statics section. Access histories for array elements are laid out the corresponding index in a dedicated shadow array. A header word in every array points to this shadow array, which is initialized lazily.

The last-write word of an access history always stores an epoch. The last-read word stores an epoch or a pointer to a read map data structure. Jikes RVM still supports x86 only in 32-bit mode, so each word is 32 bits. To distinguish epochs from pointers in the last-reads fields of access histories, epochs always have the least significant bit set to 1. We extended the garbage collector to scan the last-reads field in access histories, following the reference only if it has an least significant bit of 0. After the tag bit, the next lowest 5 bits in an epoch represent the thread identifier, and the remaining high 26 bits represent a logical clock. Representing only $2^5$ distinct threads limits the scope of executions we support but represents the best balance between avoiding clock overflow and thread identifier exhaustion in the 32-bit environment. 64-bit epochs would be less limiting.

Read maps are arrays of epochs, the same encoding used for vector clocks.[7] Given our fixed 5-bit thread identifiers, we fix vector clocks and read maps at 32 entries.

Each thread structure stores a vector clock as well as a copy of their entry in their own vector clock (the current epoch) to avoid the extra pointer dereference with this commonly used metadata.

[7]Nascent experiments with alternative read-map encodings are omitted here for lack of interesting results thus far.

Every object is a lock in Java. We lazily allocate a vector clock to track the lock's synchronization ordering the first time it is used. A reference to this vector clock is stored in a word in the header of the object. If the object is an array then this header word is used to store a reference to the shadow access history array, so the vector clock reference is stored in the header word of the shadow array for arrays. We expect the relative rarity of synchronization to make this more economical than a second header word in every object, despite the added conditional to dispatch on the type of object before retrieving the vector clock and (when locking arrays) an extra dereference. Java `volatile` fields are treated as synchronization in the Java Memory Model [76], so every volatile field has a space for vector clock reference laid out in the same object, or in the statics section for static volatile fields.

All of our data-race detector implementations maintain and rely on the revised access history invariants discussed in §5.2.2.

#### 5.5.1.2 Instrumentation

Thread fork and join operations as well as lock acquire and release operations are all treated via the normal vector-clock race-detector algorithm (§2.2.1). They require no special synchronization, since their instrumentation is always performed at program points where there is not concurrent demand for the vector clock structures involved.

Volatile field accesses have no mutual exclusion guarantees, so to avoid lost vector clock updates, we use an atomic CAS to swap a vector clock reference with a special *locked* value while using the vector clock. We have not considered applying an ownership protocol for cooperative synchronization of access to these vector clocks, but for programs with heavy use of volatile in relevant patterns, it may be helpful.

The read and write barrier fast paths (see §5.3.4 and §5.3.4.3) are inlined into application code before the accesses they monitor. The inlined barriers call out-of-line code for slow paths.

FIB uses existing yield points that are already in place in Jikes RVM for run-time services like on-stack replacement and garbage collection.

For purposes of performance evaluation, all implementations print a short report about the first data race detected in each thread, but continue execution past all detected data races. Accuracy is not guaranteed past the first data race, but the detectors continue normal execution.

### 5.5.2 FastTrack Implementations

We implemented variations on the FastTrack algorithm using the following implementations of barrier atomicity:

- SpinLock acquires a simple spin lock on the access history before *every* barrier, spinning until it succeeds at atomically comparing and swapping a special locked value into the last-reads word of an access history. After the barrier, the resulting up-to-date last-reads value is stored into the access history, simultaneously releasing the lock. Cas guarantees accuracy, albeit via a naive, pessimistic mechanism.

- Cas avoids synchronization in barriers that do not update the access history after a successful data-race check. Specifically, read barriers are synchronization-free when the access history's last-reads field holds the current epoch and when the access history's last-reads field holds a reference to a read map in which the current thread's entry is the current epoch. Write barriers are synchronization-free when the access history's last-write field holds the current epoch. Cases that update the access history require a CAS operation is required to ensure barrier atomicity, similar to the last-reads spin lock used in SpinLock. Cas guarantees accuracy via what we believe is the most effective feasible non-cooperative synchronization scheme.[8]

- Fib implements the FIB protocol discussed in §5.3, initializing access histories with they CAS-for-ownership policy (§5.4.2.1).

- Unsync uses no synchronization of its own. It makes optimizations that depend on barrier atomicity, assuming that the existing program synchronization suffices to

[8]We have also considered a refinement that allows the case matching FIB's `readSharedFirst` case to use a CAS on its individual read-map entry, rather than a CAS on the read word, plus an accompanying write-after-read-shared case that locks all read-map entries. This refinement could minimize contention in read-shared cases, but as our profiling shows in Table 5.2, these cases are too rare to have significant performance impact in practice, so we have omitted the refinement from the evaluation presented here.

provide barrier atomicity. Unsync therefore does not guarantee accuracy, but serves as a benchmark for the expected best possible performance of safe implementations of barrier atomicity.

### 5.5.3 Fib Communication Infrastructure

A thread's incoming requests and ack queues are stored as single 32-bit bit vectors in the thread's structure. As each thread may make at most one request at a time, the worst-case size of each thread's incoming request queue is linear in the number of threads. Since our epoch representation can express at most 32 unique thread identifiers (§5.5.1.1), we choose a matching queue size. The target and type of a request are stored in the requesting threads structure, since a thread has at most one outstanding request at a time. A thread may have several outstanding ack requests at a time, but they all have the same target and type. This representation makes the atomic enqueue process simpler. A requesting thread CASes out the responding thread's bit vector, replacing it with a special *locked* value. It then inspects the last-read word of the access history it is requesting and if it is still owned by the same thread, it writes back a new bit vector with the requesting thread's bit set. When a thread blocks after a yield point it CASes its queue to a special *blocked* value. Thus all state about a thread's communication is visible through this single bit vector, making atomic transitions simpler. The *locked* and *blocked* values differ per responding thread. For thread $t_i$'s queue, they use two distinct values the $i$th bit set, since thread $t_i$ never enqueues a request with itself.

Request-queue processing works as described in §5.3.5.3. Responses are stored in the requesting thread's structure. After sending a request, the requesting thread spins on its response field until the responding thread stores a response there. For acks, the single response field is used as a bit vector or counter to accumulate the set of expected responses. A requesting thread may need to process incoming requests while it is waiting for a response to its own request to avoid deadlock.

### 5.5.4 Dynamic Thread-Escape Analysis

For each version of FastTrack barrier atomicity, we implement a counterpart using dynamic thread-escape analysis as a pre-filter for access barriers and synchronization tracking. We use an implementation of a standalone conservative dynamic thread-escape analysis in Jikes RVM due to Man Cao and Mike Bond. This implementation stores an escape status in a bit of the object header, updating as described in §5.7.4. SpinLockEsc, CasEsc, FibEsc, and UnsyncEsc simply check the escape status for accesses to object fields and only continue with the barrier if the containing object is escaped. FibEscInit also does such filtering and modifies the escape analysis to store the current epoch upon escape instead of a single bit. The necessary storage space is shared with that for pointers to vector clocks and array shadows, so no additional storage is required. FibEscInit then uses the stored ownership to initialize the ownership state of fields of this object lazily. When an access barrier encounters an empty last read field of an access history, this indicates the first post-escape access to this location. The last read is derived by loading the escape epoch from the object header and Fib's protocol continues as usual. (See also §5.4.2.2.)

## 5.6 Evaluation

We evaluate the performance of Fib in two parts. First, to evaluate whether—and by how much—Fib reduces or increases the cost of barrier atomicity versus pessimistic implementations, we compare the performance of Fib against that of other data-race detectors using different implementations of barrier atomicity, but the same detection algorithm. We examine Fib's performance relative to these other implementations and in absolute terms against native execution without data-race detection. Second, to evaluate how well Fib's cooperative protocol matches costs of expected common and rare cases with real executions, we present detailed profiling results.

### 5.6.1 Environment

We ran performance and profiling experiments on a set of multithreaded Java benchmarks from the DaCapo benchmark suites [14] versions 2006-10-MR2 (eclipse6 and xalan6) and

9.12 (avrora9, jython9, luindex9, lusearch9 (fixed per [142]), pmd9, sunflow9, and xalan9), plus pjbb2005 [12], a fixed-workload version of the SPECjbb2005 benchmark [123]. Other benchmarks from DaCapo run more than 32 threads (which our prototype does not support), do not succeed on the base Jikes RVM, or lack interesting multithreaded behavior. Benchmarks run with either a fixed number of threads, or a number of threads depending on the number of available cores.

Experiments used the implementations described in §5.5, plus two other configurations: Base is Jikes RVM with no data-race detection; No Barriers adds synchronization tracking with vector clocks, but does not insert access histories or data-race checking. All configurations of Jikes RVM use the adaptive just-in-time optimizing compiler (FastAdaptive) and the generational immix garbage collector (GenImmix). Profiling experiments insert additional profiling counters in our FastTrack implementations. These counters are not present in the versions run for performance experiments. The performance and profiling results are from different sets of executions.

All experiments are run on a machine using a Linux 2.6.32 kernel on 2 × 4-core Intel Xeon E5520 CPUs at 2.26GHz with 2 SMT hardware threads per core (8 cores, 16 threads), 8MB L3 cache per CPU with 64-byte cache lines, and 10GB of RAM.

Performance experiments use 15 cores, since one benchmark (sunflow9) uses $2 \times |\text{cores}| + 1$ threads, exceeding our 32-thread limit when 16 cores are available. Scaling experiments use 2, 4, 8, 15, and 16 cores.

### 5.6.2 Performance and Profiling Results

The goal of Fib is to reduce the overhead of metadata synchronization, to help lower performance overheads for fully accurate dynamic data-race detection implemented in pure software. Motivated by data-race exceptions, we take guaranteed first-race accuracy as a requirement and good performance as a highly desirable property.

We ran each configuration of Jikes RVM described in §5.6.1 10 times on each machine configuration, collecting timing information. Figure 5.13 shows performance results on 15 of 16 cores, with execution times of each configuration normalized to those of Jikes RVM without data-race detection (Base).

Table 5.2 shows measurements of the spread of ownership state transitions FIB takes in practice, measured over a separate set of profiling runs. Rows are benchmarks and sub-rows are individual configurations of the FIB algorithm (see §5.5.2 and §5.5.4). Columns are the types of transitions, divided into two categories: **Pure** transitions examine but do not update the access history; transitions in the **Updates Access History** category update the access history. The **Not Escaped** column counts data-race checking barriers avoided using dynamic thread-escape analysis; **Local** transitions and **Fence** transitions are described in §5.3; **First (CAS)** transitions occur at the first access to a location under CAS-for-ownership initialization policy and for static field accesses in all policies (see §5.4.2); **Single-Conflict** transitions and **Multiple-Conflict** transitions are described in §5.3. Each cell contains the absolute number of transitions of this type on this benchmark under this FIB configuration, plus this number expressed as a percentage of the total number of transitions (accesses) on this benchmark under this FIB configuration (*i.e.*, over the sum of the full sub-row of absolute counts).

Figure 5.14 shows scalability results from 2 to 16 cores. Note that some benchmarks (lusearch9, sunflow9, and xalan9) derive thread counts from the number of available cores, while the rest use fixed thread counts.

### 5.6.3 Discussion

In this section we examine themes in the empirical behavior of FIB and other FastTrack implementations. Our data-race detection implementations range from 1.5× as slow (avrora9 on Unsync) to about 48× as slow (sunflow9 on SpinLock) as normal Java execution on Jikes RVM without data-race detection. (*Overheads* thus range from 50% to 4700%.) Variability is minimal except on lusearch9, which exhibits relatively high variability on all data-race detector configurations. There are several factors, from JVM to implementation level to benchmarks, that make these performance results incomparable with the average 8.5× normalized execution time reported for the original FastTrack data-race detector in [52].

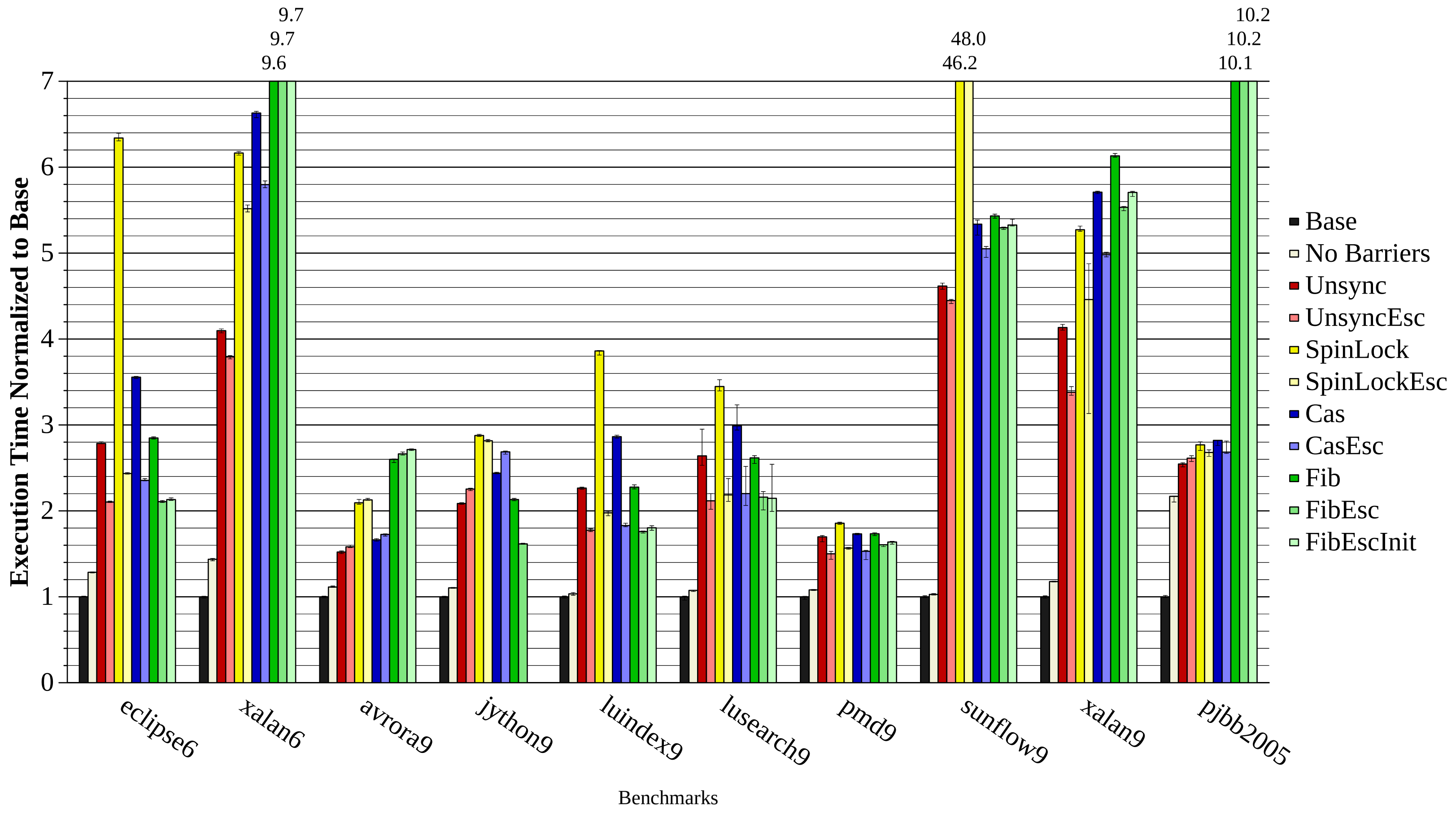
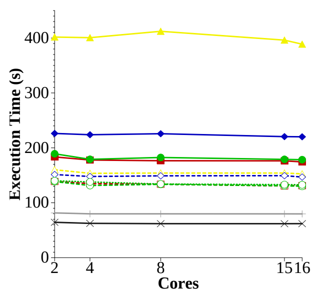


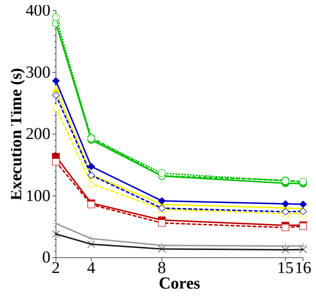
**Figure 5.13:** Execution times of data-race detector implementations normalized to Base Jikes RVM.

| | | Pure | | Updates Access History | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Not Escaped | Local | Local | Fence | First (CAS) | Single-Conflict | | Multi-Conflict |
| | | | | | | | Excl → Excl | Excl → Shared | |
| eclipse6 | Fib | 0 | 13,246,520,597 | 1,562,258,300 | 12,155 | 1,219,052,568 | 963,322 | 8,586 | 2,194 |
| | | 0% | 82.6% | 9.75% | 0.0000758% | 7.61% | 0.00601% | 0.0000536% | 0.0000137% |
| | +Esc | 13,515,037,259 | 2,045,108,099 | 363,955,331 | 7,591 | 105,991,912 | 794,895 | 3,790 | 1,289 |
| | | 84.3% | 12.8% | 2.27% | 0.0000474% | 0.661% | 0.00496% | 0.0000236% | 0.00000804% |
| | +Init | 13,510,142,210 | 2,046,674,929 | 364,350,974 | 11,786 | 104,413 | 1,114,104 | 4,811 | 814 |
| | | 84.8% | 12.9% | 2.29% | 0.000074% | 0.000656% | 0.007% | 0.0000302% | 0.00000511% |
| xalan6 | Fib | 0 | 6,198,120,224 | 6,249,879,792 | 23,399 | 701,062,430 | 150,709,494 | 1,424 | 19 |
| | | 0% | 46.6% | 47.0% | 0.000176% | 5.27% | 1.13% | 0.0000107% | $1.43 \times 10^{-7}$% |
| | +Esc | 6,130,866,197 | 2,472,848,887 | 4,296,465,186 | 21,388 | 253,985,385 | 149,925,095 | 1,238 | 18 |
| | | 46.1% | 18.6% | 32.3% | 0.000161% | 1.91% | 1.13% | 0.00000931% | $1.35 \times 10^{-7}$% |
| | +Init | 6,148,530,139 | 2,488,549,132 | 4,290,064,832 | 20,662 | 23,168 | 149,424,600 | 1,097 | 19 |
| | | 47.0% | 19.0% | 32.8% | 0.000158% | 0.000177% | 1.14% | 0.00000839% | $1.45 \times 10^{-7}$% |
| avrora9 | Fib | 0 | 7,276,621,877 | 539,552,403 | 864,301 | 45,282,123 | 35,054,978 | 4,985 | 466 |
| | | 0% | 92.1% | 6.83% | 0.0109% | 0.573% | 0.444% | 0.0000631% | 0.0000059% |
| | +Esc | 128,014,449 | 7,203,059,883 | 528,956,171 | 864,300 | 1,935,392 | 34,427,768 | 4,793 | 821 |
| | | 1.62% | 91.2% | 6.7% | 0.0109% | 0.0245% | 0.436% | 0.0000607% | 0.0000104% |
| | +Init | 128,447,408 | 7,203,939,949 | 529,386,011 | 864,033 | 233,257 | 34,564,620 | 5,066 | 1,092 |
| | | 1.63% | 91.2% | 6.7% | 0.0109% | 0.00295% | 0.438% | 0.0000641% | 0.0000138% |
| jython9 | Fib | 0 | 4,874,439,027 | 650,978,450 | 0 | 645,094,466 | 73 | 1 | 0 |
| | | 0% | 79.0% | 10.5% | 0% | 10.5% | 0.00000118% | $1.62 \times 10^{-8}$% | 0% |
| | +Esc | 4,712,194,016 | 913,226,774 | 541,735,408 | 0 | 135,330 | 27 | 2 | 0 |
| | | 76.4% | 14.8% | 8.78% | 0% | 0.00219% | $4.38 \times 10^{-7}$% | $3.24 \times 10^{-8}$% | 0% |
| | +Init | 4,722,776,592 | 923,431,761 | 573,738,434 | 0 | 48 | 27 | 2 | 0 |
| | | 75.9% | 14.8% | 9.22% | 0% | $7.72 \times 10^{-7}$% | $4.34 \times 10^{-7}$% | $3.22 \times 10^{-8}$% | 0% |
| luindex9 | Fib | 0 | 346,281,673 | 57,277,402 | 0 | 11,630,063 | 1,211 | 2 | 1 |
| | | 0% | 83.4% | 13.8% | 0% | 2.8% | 0.000292% | $4.82 \times 10^{-7}$% | $2.41 \times 10^{-7}$% |
| | +Esc | 374,518,601 | 39,641,255 | 291,791 | 0 | 32,465 | 710 | 2 | 1 |
| | | 90.4% | 9.56% | 0.0704% | 0% | 0.00783% | 0.000171% | $4.83 \times 10^{-7}$% | $2.41 \times 10^{-7}$% |
| | +Init | 375,906,265 | 39,658,652 | 271,758 | 0 | 534 | 718 | 2 | 1 |
| | | 90.4% | 9.54% | 0.0654% | 0% | 0.000128% | 0.000173% | $4.81 \times 10^{-7}$% | $2.4 \times 10^{-7}$% |
| lusearch9 | Fib | 0 | 2,590,564,430 | 365,306,939 | 2,960 | 95,502,775 | 609,175 | 79 | 0 |
| | | 0% | 84.9% | 12.0% | 0.000097% | 3.13% | 0.02% | 0.00000259% | 0% |
| | +Esc | 3,023,134,220 | 27,241,990 | 5,498,196 | 2,008 | 9,676 | 110,323 | 52 | 0 |
| | | 98.9% | 0.891% | 0.18% | 0.0000657% | 0.000317% | 0.00361% | 0.0000017% | 0% |
| | +Init | 3,048,959,312 | 31,190,388 | 55,108 | 2,005 | 7,251 | 52,928 | 44 | 1 |
| | | 99.0% | 1.01% | 0.00179% | 0.0000651% | 0.000235% | 0.00172% | 0.00000143% | $3.25 \times 10^{-8}$% |
| pmd9 | Fib | 0 | 573,181,058 | 44,929,534 | 17,532 | 58,681,794 | 1,296,562 | 757 | 32 |
| | | 0% | 84.5% | 6.63% | 0.00259% | 8.65% | 0.191% | 0.000112% | 0.00000472% |
| | +Esc | 597,984,400 | 60,620,635 | 17,301,128 | 15,373 | 2,054,123 | 1,189,281 | 1,855 | 54 |
| | | 88.0% | 8.93% | 2.55% | 0.00226% | 0.302% | 0.175% | 0.000273% | 0.00000795% |
| | +Init | 606,645,723 | 60,591,174 | 17,283,255 | 16,047 | 34,353 | 1,195,135 | 1,889 | 52 |
| | | 88.5% | 8.84% | 2.52% | 0.00234% | 0.00501% | 0.174% | 0.000275% | 0.00000758% |
| sunflow9 | Fib | 0 | 22,458,991,312 | 7,780,515 | 905,161 | 1,058,758,098 | 734,393 | 35,095 | 7 |
| | | 0% | 95.5% | 0.0331% | 0.00385% | 4.5% | 0.00312% | 0.000149% | $2.98 \times 10^{-8}$% |
| | +Esc | 10,981,938,814 | 12,540,462,585 | 4,583,243 | 899,680 | 368,066 | 1,365,931 | 30,476 | 7 |
| | | 46.7% | 53.3% | 0.0195% | 0.00382% | 0.00156% | 0.00581% | 0.00013% | $2.97 \times 10^{-8}$% |
| | +Init | 11,184,148,661 | 12,578,483,600 | 4,576,745 | 905,347 | 80,573 | 937,688 | 31,171 | 7 |
| | | 47.1% | 52.9% | 0.0193% | 0.00381% | 0.000339% | 0.00394% | 0.000131% | $2.94 \times 10^{-8}$% |
| xalan9 | Fib | 0 | 6,385,799,620 | 4,747,075,440 | 63,014 | 915,332,736 | 120,848,149 | 2,510 | 15 |
| | | 0% | 52.5% | 39.0% | 0.000518% | 7.52% | 0.993% | 0.0000206% | $1.23 \times 10^{-7}$% |
| | +Esc | 1,155,790,329 | 5,940,654,319 | 4,647,992,733 | 59,715 | 304,086,745 | 120,301,165 | 2,106 | 16 |
| | | 9.5% | 48.8% | 38.2% | 0.000491% | 2.5% | 0.989% | 0.0000173% | $1.31 \times 10^{-7}$% |
| | +Init | 1,159,743,464 | 6,106,246,634 | 4,646,292,809 | 61,479 | 28,861 | 120,318,315 | 2,148 | 17 |
| | | 9.64% | 50.7% | 38.6% | 0.000511% | 0.00024% | 1.0% | 0.0000179% | $1.41 \times 10^{-7}$% |
| pjbb2005 | Fib | 0 | 4,199,607,738 | 3,823,644,444 | 940,944 | 759,038,261 | 196,215,212 | 631,947 | 2,153,591 |
| | | 0% | 46.8% | 42.6% | 0.0105% | 8.45% | 2.18% | 0.00704% | 0.024% |
| | +Esc | 1,658,967,315 | 3,501,133,082 | 3,549,758,384 | 962,269 | 77,902,527 | 186,057,396 | 640,461 | 2,161,941 |
| | | 18.5% | 39.0% | 39.5% | 0.0107% | 0.868% | 2.07% | 0.00713% | 0.0241% |
| | +Init | 1,674,636,083 | 3,492,145,992 | 3,542,466,688 | 929,721 | 2,509,327 | 187,565,714 | 632,126 | 2,070,916 |
| | | 18.8% | 39.2% | 39.8% | 0.0104% | 0.0282% | 2.11% | 0.0071% | 0.0233% |

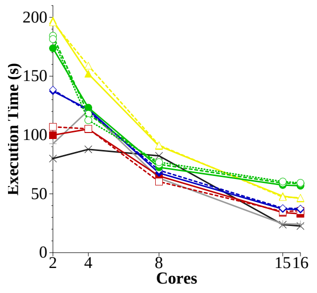
**Table 5.2:** FIB ownership state transitions in practice.



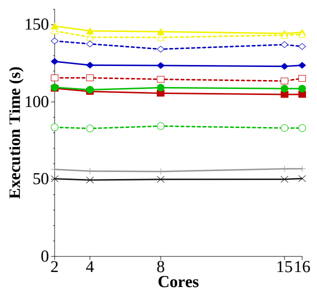
**(a)** eclipse6, 18



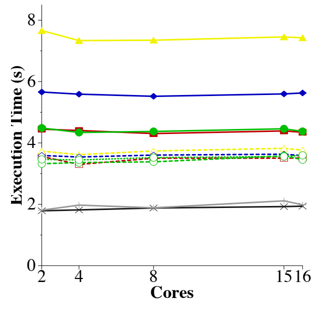
**(b)** xalan6, 9



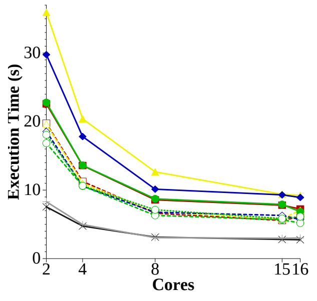
**(c)** avrora9, 27



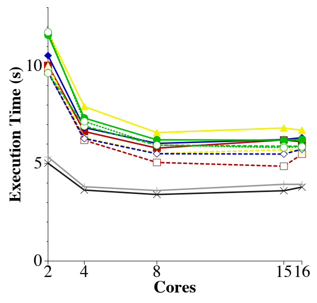
**(d)** jython9, 3



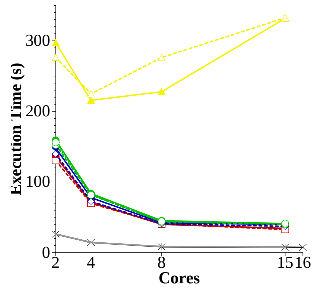
**(e)** luindex9, 2



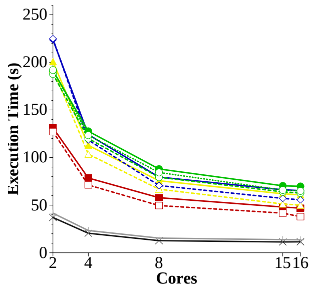
**(f)** lusearch9, |cores| + 1



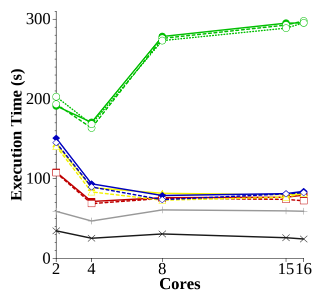
**(g)** pmd9, 5



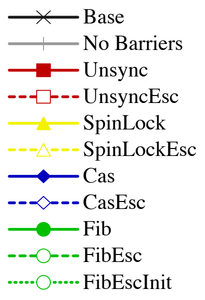
**(h)** sunflow9, 2×|cores| + 1



**(i)** xalan9, |cores| + 1



**(j)** pjbb2005, 9



**(k)** Key

**Figure 5.14:** Scalability of Base Jikes RVM and FastTrack implementations. Each benchmark is listed with the number of threads it runs in each execution.

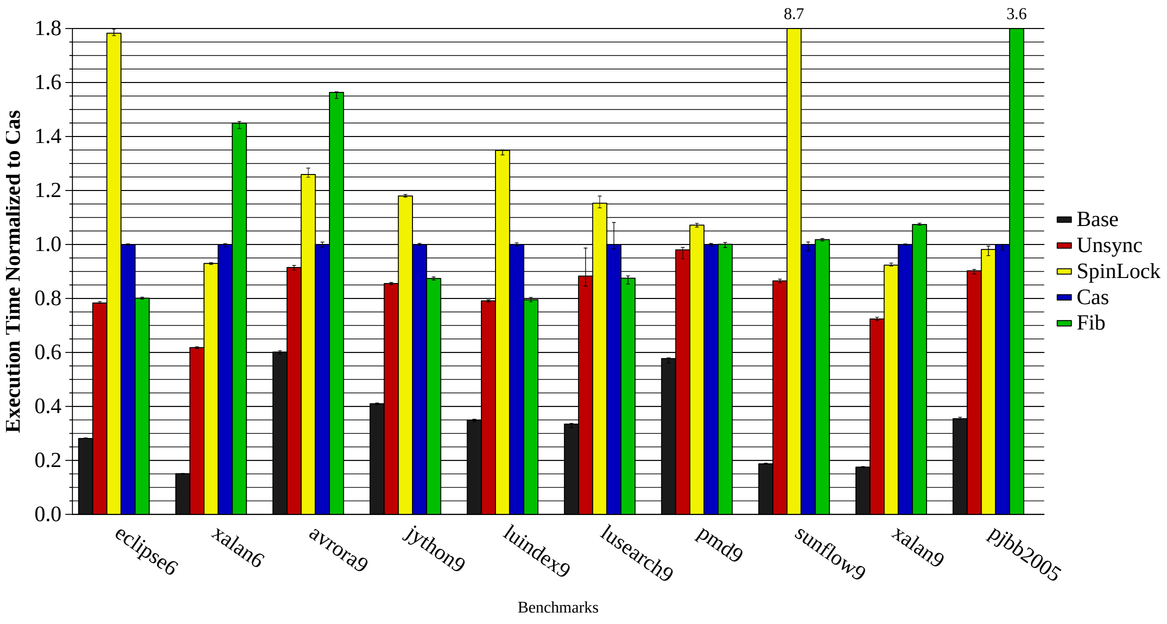


**Figure 5.15:** Execution times of Base, Unsync, Cas, and Fib, normalized to Cas.

#### 5.6.3.1 Fib versus Cas

To evaluate how well Fib achieves its primary goal of reducing overhead of metadata synchronization, we focus on the performance of Fib relative to traditionally synchronized accurate data-race detection implementations (Cas). We use an optimized unsynchronized implementation without accuracy guarantees (Unsync) as a rough proxy for the best performance we could expect from synchronization changes alone. Figure 5.15 shows a subset of the performance results from Figures 5.13 normalized to execution times of Cas, our most sophisticated pessimistic implementation of FastTrack. This presentation emphasizes two ratios. First, comparing execution times of Unsync to those of Cas shows that 2% (pmd9) to 38% (xalan6) of the full execution time of Cas is spent on synchronization.[9]

Second, comparing Fib to Cas in this plot shows that Fib runs between 21% faster (luindex9) and 260% slower (pjbb2005) than Cas. The range of performance of Fib versus

[9]Some earlier work has characterized the percentage of execution time *overheads*—*i.e.*, the time left after subtracting the execution of the unmodified JVM—are due to synchronization [38]. We always refer to percentages of full execution times rather than overheads. The percentages reported here would of course be larger if reported as overheads.

Cas is mixed. Four benchmarks (eclipse6, jython9, luindex9, lusearch9) are 13-21% faster on Fib than Cas. Furthermore, on these benchmarks, the performance of Fib is within 3% of that of the completely unsynchronized Unsync implementation. In these cases Fib delivers full accuracy guarantees with nearly no synchronization cost. Table 5.2 shows that these benchmarks have very low communication rates and high rates of synchronization-free barriers. Three benchmarks (pmd9, sunflow9, and xalan9) are as fast as or up to 7% slower on Fib than on Cas.

The remaining benchmarks are signficantly slower on Fib than on Cas: xalan6 is 45% slower, avrora9 is 57% slower, and pjbb2005 is 260% slower. Profiling results in Table 5.2 show that the these benchmarks exhibit relatively high rates of communication under Fib, with 1.1%, 0.44%, and 2.2% of all barriers requiring communication, respectively. With the exceptions of avrora9 and pmd9, all other benchmarks' communication rates are at least an order of magnitude smaller. FIB performance for avrora9 is exacerbated by the fact it runs 27 threads, but our machine offers just 15 hardware threads in these experiments. Thread preemption does not move a thread into FIB's blocking state, so a thread making a request to a descheduled thread must wait for that descheduled thread to be scheduled again and send a response, rather than self-serving the request, as would be ideal.

Neither Fib nor Cas consistently out-performs the other for all benchmarks. Fib is faster than or within 7% of Cas for 7 of 10 benchmarks and at least 45% slower for the remaining 3, while Cas is at least as fast Fib for 6 of 10 benchmarks and at least 12% slower on the remaining 4. This split suggests that selecting the synchronization scheme on a per-benchmark—or per-memory location—basis could achieve the best performance overall. We discuss the potential for adaptive techniques in §5.9.2.

#### 5.6.3.2 Dynamic Thread-Escape Analysis

Adding dynamic thread-escape analysis as a filter for data-race checking barriers and synchronization has mixed effects on performance. Revisiting Figure 5.13, escape filtering noticeably improves performance on at least one synchronized FastTrack configuration for several benchmarks. It has signficant negative performance effects in some configurations in avrora9, sunflow9, and jython9.

Not surprisingly, thread-escape filtering often has greater impact on pessimistically synchronized versions of FastTrack than on Fib, to the point that FibEsc and FibEscInit rarely perform signficantly better than CasEsc. While FibEscInit never requires synchronization on accesses to data that are thread-private, Cas must use a CAS every time it updates an access history regardless of whether it is private. For example, on eclipse6, CasEsc runs 35% faster than Cas by filtering out many data-race checks with escape status. Results for Fib and Unsync show that pessimistic synchronization is not the only overhead escape-filtering can mitigate: updates to access histories do have costs, but never affect accuracy if the access histories are not yet escaped (see §5.4.1). On eclipse6, escape-filtering to omit these synchronization-free updates in FibEscInit and UnsyncEsc causes these implementations to run about 25% faster than their counterparts without escape analysis.

Escape analysis for jython9 actually slows UnsyncEsc and CasEsc by close to 10% versus Unsync and Cas, while FibEsc runs over 20% faster than Fib and 40% faster than CasEsc in this case. Profiling information in Table 5.2 shows that Fib required CAS transitions on over 644 million accesses—over 10% of all accesses—while FibEscInit requires a CAS transition on the first access to each of 48 static fields (0.00000078% of accesses). About 99.98% of accessed memory *locations* in jython9 never escape. We discuss the relative merits of dynamic thread-escape and the Fib protocol further in §5.8.3.

#### 5.6.3.3 Scalability and Other Pessimistic Implementations

For the most part, the scalability of the data-race detectors largely follows that of the Base Jikes RVM, with proportional overheads, as seen in Figure 5.14. This trend continues on benchmarks not shown in Figure 5.14, which have nearly flat scalability graphs on all configurations. The first exception to this trend is pjbb2005 running on Fib and its variants. While other configurations show modest scaling up to 4 or 8 cores, Fib and its variants show negative scaling: the high rate of communication is exacerbated by more live threads. When most threads are blocked, as with lower core counts, many communication requests can be self-served with a couple CAS operations for synchronization in place of expensive round-trip communication waiting for yield points.

The second scaling exception is sunflow9 running on SpinLock, which acquires a lock for every barrier. Unlike Cas, SpinLock requires synchronization for all barriers. Its performance is between about 10% faster and 80% slower than Cas on all benchmarks except sunflow9, where it is about 9× as slow as Cas and 46× as slow as Base with 15 cores. sunflow9 has significant read-sharing behavior. SpinLock's locking operations on every barrier add massive amounts of cache coherence traffic where sunflow9 would experience none on a normal JVM. Figure 5.14h shows that, while other implementations have scaling that generally reduces overheads with growing core counts and scales somewhat proportionally to Base Jikes RVM, overheads and absolute execution times of SpinLock and its escape-filtered counterpart SpinLockEsc both increase with the breadth of read-sharing.

The non-starter pessimistic SpinLock implementation uses a CAS to lock the read word for *every* barrier, even in the same-epoch cases, since FastTrack's same-epoch optimizations are such obvious opportunities to avoid synchronization in the common case. It is often 10-80% slower than Cas, although it is sometimes up to 10% faster (when Cas's optimistic synchronization-free path fails frequently). SpinLock is about 790% slower than Cas on sunflow9, which has significant read-sharing. Cas benefits from a synchronization-free fast-path when a thread reads read-shared data repeatedly in the same epoch. SpinLock locks for every barrier, causing heavy write contention and resulting cache coherence transitions where the program behavior allows the program data to stay shared in all caches, with no coherence required.

We omit barrier atomicity implementations that require dedicated synchronization storage, such as a biased lock for each access history or an Octet state for each object. FastTrack is not directly expressible as an Octet client analysis (§5.7.3). We discuss potential opportunities to improve the Fib protocol with modest dedicated state storage in §5.9.

## 5.7 Related Work

Cooperative synchronization based on thread ownership has been employed in a range of settings previously. The most relevant to Fib are biased locking (§5.7.1), cache coherence and associated techniques like Radish local permissions (§5.7.2), and Octet and object granularity race detection (§5.7.3). We also discuss related work on sound dynamic thread-escape analysis (§5.7.4).

### 5.7.1 Biased Locking

Biased locking [67, 99, 113, 126] is a well-known lock implementation technique targeting locks that are acquired by one thread only, at least within some large consecutive series of acquires. Initially the lock is unowned and must be acquired in the typical fashion. A thread may bias a lock it acquires by tagging the lock with its ID. Typically, this is stored in the lock word, with another bit that indicates bias is enabled. Once a lock is biased, the bias owner thread may acquire it and release it with a single memory access and no fences, atomic compare-and-swap instructions, or other expensive operations. If another thread tries to acquire the lock, it must request that the bias owner unbias the lock or transfer bias to the requesting thread. Threads check for such requests at regular, well-defined yield points in the program. Communication to request bias changes is more expensive than normal lock operations, but also much rarer. Biasing is handled adaptively to ensure a program does not have to pay for frequent, repeated bias changes. After some number of bias changes (typically one), the lock typically enters an unbiased mode, where normal locking techniques are used to handle contention or frequent sharing.

Our algorithm differs from the simple use of a biased lock to protect each unit of metadata. In our algorithm, the bias owner for a location is derived from the access history for that location. There is no separate explicit representation of the bias owner. As a result, the metadata never enters an unbiased mode and it cannot be preemptively rebiased.

### 5.7.2 Coherence, Permissions, and Protections

MESI cache coherence [96] is a well-known hardware protocol to ensure that copies of data stored in private caches on separate cores or processors do not diverge, preserving the illusion of a single shared memory. When a processor has a cache line in sufficiently permissive ownership state, it can execute accesses to that line without communicating to other processors, but otherwise it must perform extra communication to acquire that ownership state.

The RADISH [38] hybrid software-hardware data-race detector harnesses cache coherence events to optimize its analysis. The key observation for this optimization is that after checking an access to location $x$ and finding it is data-race-free, all accesses to $x$ of the same or lesser type (read $<$ write) that occur before the next cache coherence on the cache line containing $x$ (or its eviction) are also guaranteed to be data-race free. RADISH exploits this guarantee by memoizing the result of checks as *local permissions* and downgrading these permissions as needed in response to cache coherence events. RADISH permissions are quite similar to lock bias, with potential yield points coming more frequently. While our algorithm's use of ownership information is similar in spirit to cache coherence, or its use by RADISH and some other data-race detectors [58, 82, 110] and more general analyses [85], our algorithm does not enjoy hardware support. Aikido [95] uses an extended hypervisor and virtual memory page protections for thread ownership to avoid instrumenting accesses to thread-local data.

### 5.7.3 Object Race Detection and Octet

Per-object thread ownership is used as an optimization to filter out data-race checks in von Praun and Gross's algorithm for object race detection [130]. However, as discussed in §2.5, object-granularity race detection both misses true data races and reports false data races.

OCTET [23] also associates an ownership state with each object, and reports conflicting state transitions to a client analysis. On every access to an object field, the state is checked. When the accessing thread owns the object, or if the thread is reading and the object is in a shared state, no transition is required. A conflicting transition occurs when one thread owns the object and another accesses it. To safely make this transition, the current thread must communicate with the owner thread to receive ownership and do analysis in a well-synchronized way before it can proceed. To facilitate this communication, threads check for transition requests at yield points (inserted at loop back-edges, call, return, etc.).

Our algorithm differs from OCTET mainly in that ownership state is derived from access history metadata rather than stored explicitly, meaning no extra storage is required, and state is at the granularity of fields rather than objects. While OCTET may suffer from false sharing if fields have independent ownership patterns in practice, paying many transitions for false sharing, it may also benefit if several fields of an object tend to be accessed together, paying a single object-level transition for many field-level ownership transitions.

Pure Octet could be used as a synchronization mechanism to protect data-race detection metadata, but data-race detection cannot exploit Octet's conflicting transition notifications: due to object granularity false sharing, all accesses must be still be checked by the data-race detector, regardless of whether there is a conflicting transition. (This is the same reason object race detection [130] is problematic.) Even using field-granular Octet, data-race detection must record some accesses that do not cause conflicting transitions because these accesses might become the last access before a future conflicting transition. On that (possibly racing) transition, knowledge about the last access before the transition is needed to decide if a data race occurs.

### 5.7.4 Dynamic Thread-Escape Analysis

Conservative static thread-escape analysis (see §2.6) has been used as a pre-filter to a number of thread-aware analyses that need (or less) analysis on data that are proveably thread-private. However, as a static analysis, it has precision limits. A dynamic thread-escape analysis can potentially identify more thread-private data as well as monitoring data that is initially thread-private, and making an anlysis transition when this data becomes shared. We consider two types of dynamic thread escape analysis.

The first analysis accurately identifies the first non-thread-private access, but used as a pre-filter for data race checks, it may cause missed data races. A precise dynamic thread escape analysis, such as that used in Eraser [115] or the thread-local filter tool in the RoadRunner dynamic analysis framework [54], tags each memory location with the thread that first accessed that location. On all later accesses, the analysis checks whether the accessing thread is the same as the original thread. If it is not, then the location transitions to shared. This analysis is precise, in that it initiates a transition to thread-shared mode only if (and exactly when) the location becomes accessed by multiple threads, but used as a pre-filter for data-race checks, it can lead to false or missed data races, as discussed in §2.4.

The second analysis conservatively identifies the first time a location becomes *reachable* by some thread other than its initial owner. It may identify locations as escaped and acessible by multiple threads before any non-thread-private access occurs and even may make this identification for locations that are never accessed by more than one thread. This type of analysis has been used to implement thread-local heaps [42], accelerate software transactional memory [119], and in accelerate some previous data-race detectors [33, 72, 92]. Reachability-based dynamic thread-escape analysis can serve as a safe pre-filter for data-race checks and synchronization tracking, never causing missed or false data races (see §5.4.1). However, all thee of the data-race detectors described in [33, 72, 92] may miss data races. Object race detection is employed in [92] and unsound stationary analysis is applied in [72]. TRaDE [33] starts with a thread-escape analysis that does not miss data races under the first-race guarantee, although the paper offers no justification. However, it adds a refinement optimization whereby it halts data-race detection on privatized objects, those objects that were formerly reachable by multiple threads but are now reqachable by only one thread. This optimization can miss data races that cross privatization.[10] We describe a sound use of dynamic escape status for filtering data-race-checks and synchronization tracking for FIB and other accurate vector-clock-based data-race detectors in §5.4.1. Our current implementation does not try to exploit reprivatization of data, but we discuss a sound version of refinement in §5.9.3.

## 5.8 Limitations

The basic version of FIB presented in this chapter has a number of limitations, some of which may be addressed by refinements to the protocol.

[10]Refinement is done in the garbage collector. It is unclear if the TRaDe analysis considers synchronization in the garbage collector. If it does, then the refinement optimization does not miss any data races that the TRaDe algorithm would detect without the optimization. However, as we discuss in §4.2.2, tracking such synchronization in language implementations can cause missed language-level data races.

### 5.8.1 Sensitivity to Serialized Sharing

The most significant limitation of our current Fib implementation is that its performance is highly sensitive to the rate of communication required. For programs with very high rates of temporally thread-private or read-shared data accesses, Fib excels. Its common-case synchronization barriers allow it to run noticeably faster than more pessimistic implementations. For programs with slightly lower rates of temporally private and shared accesses and higher rates of serialized sharing, the expense of Fib's heavyweight rare-case synchronization dominates, causing Fib to run significantly slower than pessimistic implementations.

These problematic sharing patterns typically correlate with performance bottlenecks such as poor cache behavior even in uninstrumented applications. In many cases, while Fib has poor performance on these sharing patterns, it arguably exacerbates an existing problem rather than introducing overheads where application performance was otherwise well-optimized. Nonetheless, Fib can add signficant overheads even for moderates of such sharing. This sensitivity is likely due to several factors.

First, as Fib derives the ownership state of a location purely from its access history metadata, Fib state transitions cannot be decoupled from data-race detection analysis updates. Second, Fib maintains a state per location, meaning large sets of locations accessed in the same pattern require individual state transitions. These two features prevent preemptive or bulk transitions that would be possible in other more decoupled or coarser-grained ownership tracking systems.

Third, Fib's limited states (Exclusive or Shared), also tied to the pure state derivation feature, force expensive communication on some transitions that are cheaper in other ownership tracking systems that offer more states to represent more refined access permissions and transitions.

#### 5.8.1.1 Fib versus Octet

Fib's three most problematic benchmarks (xalan6, avrora9, and pjbb2005) are also the benchmarks with the highest performance overheads and highest rates of communication for Octet [23]. Fib's rates of communication and overheads versus Unsync follow the same pattern as Octet's, but are proportionally higher. While this is not a directly meaningful comparison, it roughly measures the overhead of the synchronization protocol alone by using the unsynchronized data-race detector as a baseline for Fib. There are two protocol differences that may contribute the higher relative overheads of Fib.

First, Octet associates ownership state with objects, not individual fields and array elements. Thus Octet's states and transitions experience a false sharing effect with potential detriments and benefits. When fields of the same object or elements of the same array have independent thread ownership, Octet's object-granular states can cause falsely conflicting transitions and incur unneeded communication overhead. A pattern on which this would occur is an array with one element per thread, where each thread's element is treated as thread-private to that thread. In the worst case for Octet, Octet could incur communication on every access where Fib incurs no synchronization at all. However, when multiple fields of a single object or multiple elements of a single array are all accessed together by the same thread in the same pattern exhibiting spatial thread-locality, Octet's object-granular states amortize the cost of communication for an access to the first field of an object over accesses to several fields in that object. In the worst case, Fib could incur communication for every individual field accessed.

Second, Octet breaks Fib's Exclusive state into two states, WrEx and RdEx, allowing for some synchronization-free transitions that would require communication in Fib, at the cost of CAS transitions on some cases where Fib is synchronization-free. Specifically, Octet allows a RdEx $\rightarrow$ RdSh transition with a single CAS, while Fib requires communication for Exclusive $\rightarrow$ Shared. (See the discussion of such transitions in §5.3.5.3.) On the other hand, Fib allows writes following reads as Exclusive($t$) $\rightarrow$ Exclusive($t$) transitions with no synchronization, while Octet requires a CAS for upgrades from read to write permission: $\mathsf{RdEx}_\mathsf{T} \rightarrow \mathsf{WrEx}_\mathsf{T}$. While Fib can be extended to use WrEx and RdEx states, they are more restrictive than those in Octet (see §5.9.1).

Furthermore, Fib's policy of deriving state purely from FastTrack access history metadata forces Fib to follow FastTrack's policies on metadata update. Specifically, FastTrack's optimization for globally-ordered reads avoids storing a full read map when reads are ordered by locking, for example. This allows later write checks to succeed with constant-time checks rather than time linear in the number of threads. This also forces Fib to make some Exclusive($t$) → Exclusive($u$) transitions—each requiring communication—where Octet could make a transition to RdSh, allowing future read barrier to complete free of synchronization.

Fib's Exclusive → Shared transition (see Table 5.2) is rarer than Octet's RdEx → RdSh transition (see Table 3 in [23]) in practice, suggesting Fib may be negatively impacted by FastTrack's globally-ordered reads optimization. However, Octet's RdEx → RdSh transition is infrequent enough that this tradeoff does not likely have significant impact on performance. Besides a margin for the more carefully tuned concrete communication implementation in Octet, this leaves the conclusion that object granularity likely helps Octet versus Fib overall. This suggests a possible limit on further optimizations of Fib, since object granularity is poorly suited to accurate data-race detection, as discussed in §5.7.3.

### 5.8.2 Starvation of Atomic Lookup-and-Enqueue

The three limitations of the basic Fib protocol that contribute to its sensitive performance also contribute to the theoretical starvation of atomic lookup-and-enqueue operations discussed in §5.3.5.1. In practice, starvation never occurred in our experiments, and even single failed atomic lookup-and-enqueue operations were rare. In the worst case, pjbb2005 retried 1253 failed atomic lookup-and-enqueue operations in a single execution, at a rate of 1 in 7.2 million accesses or 1 in 150,000 requests. In all other benchmarks, no more than 22 atomic lookup-and-enqueue operations required retry, with many benchmarks never experiencing failed atomic lookup-and-enqueue.

Distinct Fib state for each location could mitigate the possibility of starvation by replacing the two-step lookup-and-enqueue operation with a single CAS operation. We have not investigated this option in detail given the lack of observed starvation in practice.

### 5.8.3 Dynamic Thread-Escape Analysis

The Fib protocol is naturally optimized to make analysis of thread-private data relatively inexpensive. While dynamic thread-escape filtering can reduce Fib's overheads further for thread-private data, other more pessimistic synchronization implementations see much greater reductions in overheads due to dynamic thread-escape filtering, as it eliminates common-case overheads in much the same way that the FIB protocol does. With dynamic thread-escape analysis, the performance gains of FIB are less significant. On the one hand, this suggests that FIB is natural fit for many programs and mitigates the need for thread-escape information. On the other hand, it suggests that—in practice on the benchmarks we evaluated—the version of the FIB protocol implemented here does not offer significant gains over a simple synchronization scheme combined with a simple dynamic thread-escape analysis. We discuss possible improvements to FIB in §5.9.

## 5.9 Future Work

In this section we outline a number of future extensions to FIB to address its limitations and improve its performance.

### 5.9.1 Refined Ownership States

By adding modest explicit storage for ownership state in each access history, FIB could make several protocol variations such as distinguishing between Write-Exclusive and Read-Exclusive states (§5.9.1.1), adding intermediate states to simplify atomic lookup-and-enqueue (§5.8.2, §5.3.5.1), or enabling preemptive or bulk state transitions that do not occur hand-in-hand with data-race detection updates (§5.9.1.2).

#### 5.9.1.1 Write- and Read-Exclusive States

Splitting FIB's Exclusive state into distinct Write-Exclusive and Read-Exclusive states would realign synchronization costs closer to those of OCTET (§5.8.1.1). Unlike OCTET's WrEx and RdEx states, which refer to writing and reading of the associated application data, FIB Write-Exclusive and Read-Exclusive states would refer to writing and reading of the *access history*, since FIB's goal is to order analysis accesses, not just application accesses. As such, FIB would utilize Read-Exclusive less often than OCTET uses RdEx, since some data-race detection barriers for program read accesses require (Write-Exclusive) updates to the access history. Specifically, the FIB permissions would be as follows:

- Write-Exclusive($t$) grants thread $t$ exclusive permission for both write and read barriers to check against and update all contents of this access history without synchronization.

- Read-Exclusive($t$) grants thread $t$ exclusive permission for both write and read barriers to check against *but not update* this access history without synchronization.

Permission granted by Shared states remain the same. The distinction between Write-Exclusive and Read-Exclusive would be encoded by stealing another bit from the last-read word.

Transitions from Read-Exclusive($t$) to Write-Exclusive($t$) would require a CAS of the last-read word in the access history.[11] This transition replaces a synchronization-free Exclusive($t$) $\rightarrow$ Exclusive($t$) transition in FIB. Transitions from Write-Exclusive to Shared are the same as Exclusive $\rightarrow$ Shared transitions in FIB, but Read-Exclusive($t$) $\rightarrow$ Shared transitions are made safely with a CAS and no communication, since a barrier by thread $t$ cannot update the access history without a CAS transition to Write-Exclusive($t$).

This version of the protocol trades the added cost of a CAS on some previous synchronization-free Exclusive($t$) $\rightarrow$ Exclusive($t$) self-transitions for the elimination of communication on some previously expensive Exclusive $\rightarrow$ Shared transitions. As noted in §5.8.1.1, the rates of these transitions do not appear likely to have signficant performance impact, but we believe this protocol would still be valuable to implement and test.

#### 5.9.1.2 Independent or Bulk Transitions

A more attractive use of additional explicit state storage would enable transitions independent of data-race detection updates to the access history. By stealing an extra bit of access history, we could encode that state lookup should be either by the current FIB derivation or indirected to a word in the object header.[12] This state indirection would essentially allow for the choice between states covering data at field granularity or object granularity, while also allowing transitions independent from data-race detection metadata updates at the cost of

[11]With careful layout and manipulation of the bit fields of each word in the access history, some Read-Exclusive($t$) $\rightarrow$ Write-Exclusive($t$) transitions could be made by individual byte updates without a CAS.

[12]For further flexibility at the cost of more space, we could add a word to each access history to hold a pointer to any arbitrary state storage. See [44] for a similar construct with software transactional memory.

added time and space for state lookup. As discussed in §5.8.1.1, object granularity states are attractive when all fields of an object (or all elements of an array) should change ownership states together, as in a producer-consumer thread pipeline, for example.

### 5.9.2 Adaptive Synchronization Selection

The mixed performance of Fib versus more pessimistic implementations of barrier atomicity (see §5.6.3.1) suggests that selecting a synchronization scheme via profiling or programmer directive could offer the best overall performance. Selecting synchronization at the granularity of a single access history (or even a single phase of execution for a single access history) could offer better performance than any single synchronization scheme per application. A simple adaptive approach would associate a saturating counter with each access history or object to count the number of expensive Fib transitions it takes. Once the counter is saturated, this access history or object switches to a pessimistic barrier atomicity implementation. When combined with dynamic escape analysis, it may prove most effective to select pessimistic synchronization preemptively for an object if that object escapes via publication to an object where pessimistic synchronization is already in effect.

Cao, *et al.*, have explored more complicated combined state transition machines using both optimistic and pessimistic states for an adaptive version of Octet [75] that might inform a similar design for Fib if a simple version does not suffice.

### 5.9.3 Reprivatization for Data-Race Detection

Marking escaped objects non-escaped when they are no longer reachable by multiple threads could allow dynamic thread-escape analysis to cut performance overhead more aggressively by filtering data-race checks even for objects that were shared but have since been privatized. Privatization appears in many programming patterns, such as thread pipelining or producer-consumer patterns.

| **Thread $t_1$** | **Thread $t_2$** |
|---|---|
| let x = g | |
| release(m) | |
| | acquire(m) |
| | g.f = 2 |
| | g = null |
| — *reprivatize $o$* — | |
| let y = x.f | |

**Figure 5.16:** Unsound reprivatization in TRaDe may miss true data races. The variables g and m are global; x and y are local. Initially, g is the only reference to object $o$. Since g is a global variable, $o$ is escaped. The red accesses to $o$.f form a data race spanning privatization.

#### 5.9.3.1 Unsound Reprivatization

The TRaDE algorithm for data-race detection [33], discussed in §2.2.1.3 and §5.7.4, makes an unsound version of this optimization, called *refinement*. The garbage collector scans for newly-private objects at each collection, clears their escaped state, and erases their access histories, so future accesses to these objects will not require a data-race check until they escape again. TRaDe refinement can miss true data races that span privatization. Unlike a true escape-spanning data race that can be missed under conservative dynamic thread-escape filtering, a true privatization-spanning data races is not necessarily dominated by another true data race that TRaDe will detect.

Consider the example in Figure 5.16, where a true data race on field f of object $o$ will be missed under TRaDe's refinement. Lock m is intended to protect global variable g. Thread $t_1$ takes a local reference to $o$, then passes lock m to $t_2$, which writes to field f of object $o$ and then nulls the reference g, such that thread $t_1$'s local variable x is now the only reference to $o$. Garbage collection and TRaDe refinement occur at this point, erasing access history for $o$.f and marking $o$ as non-escaped. When thread $t_1$ reads $o$.f through its local reference x, the data-race detector will see that $o$ is non-escaped and will skip the data-race check, missing a data race between this access and thread $t_2$'s earlier access to $o$.f. This is the only data-race in the execution. It is not preceded by any other data race. Specifically, the privatizing access g = null in thread $t_2$ is not involved in a data race.

#### 5.9.3.2 Sound Reprivatization

In this example, it is not safe to mark $o$ non-escaped and erase its access history until the next synchronization from thread $t_2$ to thread $t_1$. A more general sound policy is to mark a privatized object with a third escape state, *privatized*. The dynamic thread-escape analysis treats a *privatized* object as *non-escaped*. Filtering of data-race checks treats a *privatized* object as *escaped*. The dynamic thread-escape analysis must mark a *privatized* object *escaped* (and perform accompanying transitions) if it installs an escaped reference to that object. Accesses to a field of an object in *privatized* state require data-race checks. Once a field has been read without a data race, no further read data-race checks are required on that field unless the object's state changes. Once a field has been written without a data race, the access history of that field can be erased and no further data-race checks are required on that field unless the object's state changes. Once all fields of the object have been written without a data race, the object can be marked *non-escaped* once more. More preemptive versions of this reprivatization checking are also possible.

The downside of reprivatization is the potential for repeated escape and reprivatization. If a connected component of the heap is privatized, we would like to benefit from cheaper analysis barriers on all objects in this component, so the transitive propagation done at escape should also happen at privatization. In conventional escape analysis without reprivatization, each object is marked escaped at most once. The potentially high cost of transitive escape on a single reference update is amortized. When reprivatization is introduced, there is no bound on the number of times an object may escape (or be reprivatized). In the worst case, the entire heap is a single connected component rooted in one object that repeatedly escapes and is reprivatized. At every step, a full transitive closure is required.

Measuring behavior of escape and reprivatization will be important to determine the practical impact of these differing bounds. To avoid worst-case behavior, a saturating counter can be associated with the escape status of each object. If the object has been privatized more than $k$ times, it should remain escaped permanently.

## 5.10 Conclusions

Fib is a cooperative synchronization protocol to preserve the integrity of dynamic data-race detection metadata with no synchronization in the common case and expensive synchronization in the rare case. Our evaluation shows that Fib's performance is promising, but sensitive to the rate of tight sharing of data. Fib out-performs conventional analysis synchronization techniques on some benchmarks while running slower on others. These results suggest that future work on an adaptive approach and other refinements to the protocol could achieve the best overall performance.

Chapter 6

# Conclusions and Next Steps

This dissertation described three techniques to bring data-race exceptions closer to feasibility in programming languages at various levels of abstraction. Data-race exceptions would mitigate the ill effects of data races by making every data race an explicit fail-stop error at run-time. Implementing data-race exceptions demands accurate and fast dynamic data-race detection support, yet accuracy and performance have historically been at odds in data-race detection. This dissertation showed that: a hybrid software-hardware data-race detector can optimize common cases with hardware support while handling rare cases in software to maintain full accuracy for ISA-level programs (Chapter 3); low-level data-race detectors are inaccurate for higher-level programming languages but can be virtualized to support accurate language-level data-race detection (Chapter 4); and cooperative analysis synchronization has the potential to reduce overheads of accurate data-race detection in pure software implementations we can run today (Chapter 5).

## 6.1 Summary of Conclusions

A repeated theme in this dissertation is the importance and power of designing dynamic analysis systems with careful consideration of the interaction between software and hardware, as well as their respective strengths and weaknesses.

Our design for Radish and our proof of its correctness demonstrated the value of starting with a canonical, accurate software algorithm for dynamic data-race detection and mapping performance-critical common cases into hardware. The software baseline allowed Radish to maintain full accuracy even in rare cases that are difficult to handle in hardware alone. The design process of *correctness first, performance later* resulted in a design that exploits the strengths, and mitigates the weaknesses, of both hardware and software analysis techniques. Compared to hardware data-race detectors, RADISH eliminates all sources of inaccuracy. Compared to software data-race detectors, RADISH mitigates most sources of inefficiency.

Our work with LARD evolved this co-design further, taking into account the effects of translation between execution abstractions to virtualize lower-level data-race detection resources such that they can be applied accurately to arbitrary higher-level shared-memory multithreaded execution abstractions that run via translation to a lower-level shared-memory multithreading implementation. Beyond designing a single software-hardware platform for accurate high-level data-race detection, this work synthesized a principled taxonomy of how program translation affects artifacts defined in terms of a single execution abstraction and a design pattern for translating these artifacts, drawing on fresh insights as well as *ad hoc* workarounds familiar to some dynamic analysis implementers, but not widely understood in the broader community. Furthermore, our evaluation demonstrated that translation effects are a prohibitive problem for naïve hardware-level detection of language-level data races in practice, validating the necessity of designing mutually aware software and hardware systems.

Our design and implementation of the FIB protocol eliminated hardware support in favor of pure-software data-race detection that we can evaluate and use in real-world conditions without waiting years for robust hardware implementations. Despite a lack of hardware integration, the idea for FIB arose from lessons learned designing hybrid software-hardware systems. FIB's synchronization-free optimization of common cases in a software data-race detector emulates similar optimizations accomplished via hardware support in RADISH. The initial implementation and evaluation in this dissertation demonstrate that this approach shows promise for reducing the overheads of pure-software data-race detection, although further work is necessary to determine its practical value.

Our combined work on RADISH, LARD, and FIB has demonstrated the power of mutually aware software and hardware techniques to reduce the overheads of accurate dynamic analysis and bring data-race exceptions closer to feasibility.

## 6.2 Racing Onward: Limitations and Future Work

Despite Radish's overall good performance (detailed in [38]), some benchmarks still experience overheads that are too high for most deployment scenarios. Furthermore, Radish remains a relatively high-level design. While we have designed solutions to a number of systems issues (detailed partly in Chapter 3 and more extensively in [38]), their practicality and feasibility have not been tested in the real world.

A consistent challenge in moving new hardware designs from proposal to widespread implementation is demonstrating significant utility. While we envision data-race exceptions as always-on—not just a testing or debugging feature—and highly useful, additional uses of the Radish mechanism could broaden its appeal. Our earlier consideration of constructive *misuses* of accurate and fast hardware data-race exception support generated promising ideas [136]. (In fact, it was in the course of this brainstorming that the need for Lard first became apparent.) Nonetheless, the Radish hardware mechanism is clearly purpose-built for data-race detection. A more flexible hardware mechanism that could express other similar analyses while maintaining good performance for data-race detection would broaden the appeal of Radish and its chances for adoption.

Similarly, the Radish hardware extensions required for Lardish (§4.3.2) have not been deeply vetted, and may benefit from small but fundamental revisions in a redesigned hardware data-race detector that takes abstractable data-race detection as its first goal. In particular, the merging of a modern language virtual machine with the software component of a hybrid software-hardware data-race detector should present additional optimization opportunities, but is not fully feasible in our current designs of Lardish and Jikes LardVM (§4.3).

We believe that the principles of Lard generalize beyond hardware-supported detection of language-level data races. First, the taxonomy of translation effects and the associated design pattern generalize to other pairs of execution and memory abstractions, including hypervisors, operating systems, and programming models implemented atop high-level languages. Second, we hope the design of Lard can inform that of other language-level analyses using low-level resources, such as atomicity violation detection [56] or inter-thread communication checking [138]. Yet more generally, the development and presentation of

LARD has raised many interesting questions about how programmers or analyses define and reason about notions like data races or memory locations on layered execution abstractions that do not hide all implementation details perfectly. We hope future studies will generalize our insights and develop deeper principles for this type of analysis and execution abstraction.

The evaluation of FIB presented in this dissertation demonstrated that no single software approach to barrier atomicity is most efficient in all cases (§5.6.3.1) and that performance improvements due to the current version of the FIB protocol have significant overlap with those enabled by conservative dynamic thread-escape analysis (§5.8.3). An adaptive and generally more flexible version of FIB could offer stronger performance, as discussed in §5.9. Nonetheless, this design process has opened up more general questions about how best to co-optimize a given pair of dynamic analysis and application. We foresee opportunity in steering automated optimization attempts via a combination of static analysis, dynamic profiling, adaptive protocols, and flexible optimization hints from programmers. For programmers who need both the safety of data-race exceptions and good performance, integrating moderated control of the data-race detection analysis with language or library support may enable the programmer to better co-optimize analysis and application.

Finally, the three contributions of this dissertation represent steps forward in the implementation of data-race exceptions, but their potential utility is currently limited to traditional shared-memory multithreading on a single machine. Adapting data-race exceptions and the supporting analysis techniques developed in this dissertation to a broader range of related programming models could increase their impact and offer new insights into analysis implementation.

# References

[1] Martín Abadi, Cormac Flanagan, and Stephen N. Freund. Types for Safe Locking: Static Race Detection for Java. *TOPLAS*, 28(2), 2006.

[2] Sarita V. Adve and Hans-Juergen Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53, August 2010.

[3] Sarita V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, 1996.

[4] Sarita V. Adve and Mark D. Hill. Weak Ordering—A New Definition. In *ISCA*, 1990.

[5] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *ISCA*, 1991.

[6] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan Eggers. Eliminating Unnecessary Synchronization from Java Programs. In *SAS*, 1999.

[7] T. R. Allen and D. A. Padua. "debugging fortran on a shared memory machine". In *ICPP*, 1987.

[8] Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen E. Smith. Implementing Jalapeño in Java. In *OOPSLA*, 1999. http://www.jikesrvm.org.

[9] David Bacon, Robert Strom, and Ashis Tarafdar. Guava: A Dialect of Java Without Data Races. In *OOPSLA*, 2000.

[10] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A Theory of Data Race Detection. In *PADTAD*, 2006.

[11] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.

[12] Stephen M. Blackburn. pjbb2005. `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`.

[13] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE*, 2004.

[14] Stephen M. Blackburn, Robin Garner, Chris Hoffman, Asiad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, 2006.

[15] Guy Blelloch. NESL: A Nested Data-Parallel Language. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[16] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[17] Robert L. Bocchino, Jr., Vikram Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[18] Hans-Juergen Boehm. Threads Cannot Be Implemented As a Library. In *PLDI*, 2005.

[19] Hans-Juergen Boehm. How to Miscompile Programs with ”Benign” Data Races. In *HotPar*, 2011.

[20] Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.

[21] Hans-Juergen Boehm and Sarita V. Adve. You Don’t Know Jack About Shared Variables or Memory Models. *CACM*, 55(2):48–54, February 2012.

[22] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional Detection of Data Races. In *PLDI*, 2010.

[23] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *OOPSLA*, 2013.

[24] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.

[25] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.

[26] C++ Standards Comittee, Stefanus Du Toit, *ed.* Working Draft, Standard for Programming Language C++. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf.

[27] Luis Ceze, Joseph Devietti, Brandon Lucia, and Shaz Qadeer. A Case for System Support for Concurrency Exceptions. In *HotPar*, 2009.

[28] Guang-Ien Cheng, Mingdong Feng, Charles Leiserson, Keith Randall, and Andrew Stark. Detecting Data Races in Cilk Programs that Use Locks. In *SPAA*, 1998.

[29] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape Analysis for Java. In *OOPSLA*, 1999.

[30] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, 2002.

[31] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for Debugging Parallel Programs with Flowback Analysis. *TOPLAS*, 13(4), October 1991.

[32] Jong-Deok Choi and Sang Lyul Min. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. In *PPoPP*, 1991.

[33] Mark Christaens and Koen De Bosschere. A Topological Approach to On-the-fly Race Detection in Java Programs. In *Symposium on Java Virtual Machine Research and Technology*, 2001.

[34] Maria Christakis and Konstantinos Sagonas. Static Detection of Race Conditions in Erlang. In *PADL*, 2010.

[35] Mark Christiaens and Koen De Bosschere. Accordion Clocks: Logical Clocks for Data Race Detection. In *EuroPar*, 2001.

[36] Joseph Devietti, Colin Blundell, Milo Martin, and Steve Zdancewic. HardBound: Architectural Support for Spatial Safety of the C Programming Language. In *ASPLOS*, 2008.

[37] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[38] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *ISCA*, 2012.

[39] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. Technical report, UW-CSE-12-04-01, 2012.

[40] Anne Dinning and Edith Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *PPoPP*, 1990.

[41] Anne Dinning and Edith Schonberg. Detecting Access Anomalies in Programs with Critical Sections. In *Workshop on Parallel and Distributed Debugging*, 1991.

[42] Damien Doligez and Xavier Leroy. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *POPL*, 1993.

[43] Laura Effinger-Dean, Hans-Juergen Boehm, Dhrova Chakrabarti, and Pramod Joisha. Extended Sequential Reasoning for Data-Race-Free Programs. In *MSPC*, 2011.

[44] Laura Effinger-Dean and Dan Grossman. Region-Based Dynamic Separation for STM Haskell. In *TRANSACT*, 2011.

[45] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-Juergen Boehm. IFRit: Interference-free Regions for Dynamic Data-Race Detection. In *OOPSLA*, 2012.

[46] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, 2007.

[47] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, 2003.

[48] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, 2010.

[49] Colin Fidge. Logical Time in Distributed Computing Systems. *Computer*, 24, August 1991.

[50] Stephen J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *CGO*, 2003.

[51] Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000.

[52] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[53] Cormac Flanagan and Stephen N. Freund. Adversarial Memory for Detecting Destructive Races. In *PLDI*, 2010.

[54] Cormac Flanagan and Stephen N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*, 2010.

[55] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant Check Elimination For Dynamic Race Detectors. In *ECOOP*, 2013.

[56] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A Sound And Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, 2008.

[57] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI*, 2003.

[58] Rodrigo Gonzalez-Alberquilla, Karin Strauss, Luis Ceze, and Luis Piñuel. Accelerating Data Race Detection with Minimal Hardware Support. In *ICPP*, 2011.

[59] Dan Grossman. Type-Safe Multithreading in Cyclone. In *TLDI*, 2003.

[60] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race Checking by Context Inference. In *PLDI*, 2004.

[61] Intel. Thread checker. http://software.intel.com/en-us/articles/intel-thread-checker/, 2011.

[62] Intel. Inpector XE. http://software.intel.com/en-us/intel-inspector-xe, 2013.

[63] Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. Toward Integration of Data Race Detection in DSM Systems. *Journal of Parallel and Distributed Computing*, 59(2):180 – 203, 1999.

[64] Huafeng Jin, Tuba Yavuz-Kahveci, and Beverly A. Sanders. Java Memory Model-Aware Model Checking. *TACAS*, 7214, 2012.

[65] Baris Kasikci, Cristian Zamfir, and George Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, 2012.

[66] Baris Kasikci, Cristian Zamfir, and George Candea. RaceMob: Crowdsourced Data Race Detection. In *SOSP*, 2013.

[67] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *OOPSLA*, 2002.

[68] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21, July 1978.

[69] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[70] Doug Lea. The JSR-133 Cookbook. http://g.oswego.edu/dl/jmm/cookbook.html, Retrieved January 20, 2014.

[71] Kyungwoo Lee and Samuel P. Midkiff. A Two-phase Escape Analysis for Parallel Java Programs. In *PACT*, 2006.

[72] Du Li, Witawas Srisa-an, and Matthew B. Dwyer. SOS: Saving Time in Dynamic Race Detection with Stationary Analysis. In *OOPSLA*, 2011.

[73] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-Juergen Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, 2010.

[74] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. PIN: Building Customized Program Analysis Tools With Dynamic Instrumentation. In *PLDI*, 2005. http://www.pintool.org.

[75] Man Cao and Minjia Zhang and Michael D. Bond. Drinking from Both Glasses: Adaptively Combining Pessimistic and Optimistic Synchronization for Efficient Parallel Runtime Support. In *WoDet*, 2014.

[76] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL*, 2005.

[77] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, 2009.

[78] Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, 2010.

[79] Nicholas D. Matsakis and Thomas R. Gross. A Time-Aware Type System for Data-Race Protection and Guaranteed Initialization. In *OOPSLA*, 2010.

[80] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[81] John Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *SC*, 1991.

[82] Sang L. Min and Jong-Deok Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *ASPLOS*, 1991.

[83] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, 2009.

[84] Arndt Mhlenfeld and Franz Wotawa. Fault Detection in Multi-Threaded C++ Server Applications. *Electronic Notes in Theoretical Computer Science*, 174(9):5–22, 2007.

[85] Vijay Nagarajan and Rajiv Gupta. ECMon: Exposing Cache Events for Monitoring. In *ISCA*, 2009.

[86] Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. In *PLDI*, 2006.

[87] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*, 2007.

[88] Robert H. B. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs.* PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, August 1991.

[89] Robert H. B. Netzer, Timothy W. Brennan, and Suresh K. Damodaran-Kamal. Debugging Race Conditions in Message-Passing Programs. In *SPDT*, 1996.

[90] Robert H. B. Netzer and Barton P. Miller. Improving the Accuracy of Data Race Detection. In *PPoPP*, 1991.

[91] Robert H. B. Netzer and Barton P. Miller. What Are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):7488, March 1992.

[92] Hiroyasu Nishiyama. Detecting Data Races Using Dynamic Escape Analysis Based on Read Barrier. In *Conference on Virtual Machine Research And Technology Symposium*, 2004.

[93] Robert O'Callahan and Jong-Deok Choi. Hybrid Dynamic Data Race Detection. In *PPoPP*, 2003.

[94] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[95] Marek Olszewski, Qin Zhao, David Koh, Jason Ansel, and Saman Amarasinghe. Aikido: Accelerating Shared Data Dynamic Analyses. In *ASPLOS*, 2012.

[96] Mark S. Papamarcos and Janak H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ISCA*, 1984.

[97] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. Efficient Data Race Detection for Distributed Memory Parallel Programs. In *SC*, 2011.

[98] Chang Seo Park, Koushik Sen, and Costin Iancu. Scaling Data Race Detection for Partitioned Global Address Space Programs. In *SC*, 2013.

[99] Filip Pizlo, Daniel Frampton, and Antony L. Hosking. Fine-grained Adaptive Biased Locking. In *PPPJ*, 2011.

[100] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A Study of Concurrent Real-Time Garbage Collectors. In *PLDI*, 2008.

[101] Eli Pozniansky and Assaf Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *PPoPP*, 2003.

[102] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, 2006.

[103] Milos Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In *HPCA*, 2006.

[104] Milos Prvulovic and Josep Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *ISCA*, 2003.

[105] Yao Qi, Raja Das, Zhi Da Luo, and Martin Trotter. MulticoreSDK: A Practical and Efficient Data Race Detector for Real-World Applications. In *PADTAD*, 2009.

[106] Cosmin Radoi and Danny Dig. Practical Static Race Detection for Java Parallel Loops. In *ISSTA*, 2013.

[107] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *ISCA*, 2005.

[108] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and Tolerating Asymmetric Races. In *PPoPP*, 2009.

[109] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective Race Detection for Event-Driven Programs. In *OOPSLA*, 2013.

[110] Brad Richards and James R. Larus. Protocol-Based Data-Race Detection. In *SPDT*, 1998.

[111] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *TOCS*, 17(2):133–152, May 1999.

[112] Erik Ruf. Effective Synchronization Removal for Java. In *PLDI*, 2000.

[113] K. Russell and D. Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *OOPSLA*, 2006.

[114] Alexandru Salcianu and Martin Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *PPoPP*, 2001.

[115] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *TOCS*, 15(4), 1997.

[116] Edith Schonberg. On-the-fly Detection of Access Anomalies. In *PLDI*, 1989.

[117] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[118] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. RACEZ: a Lightweight and Non-Invasive Race Detection Tool for Production Applications. In *ICSE*, 2011.

[119] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *PLDI*, 2007.

[120] Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd D. Millstein, and Madanlal Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, 2011.

[121] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound Predictive Race Detection in Polynomial Time. In *POPL*, 2012.

[122] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC*, 2001.

[123] Standard Performance Evaluation Corporation. SPECjbb2005. `http://www.spec.org/jbb2005/`.

[124] Tachio Terauchi. Checking Race Freedom Via Linear Programming. In *PLDI*, 2008.

[125] Valgrind Project. Helgrind: a thread error detector. `http://valgrind.org/docs/manual/hg-manual.html`, 2013.

[126] Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. Simple and Fast Biased Locks. In *PACT*, 2010.

[127] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, 2011.

[128] Kaushik Veeraraghavan. *Uniparallel Execution and Its Uses.* PhD thesis, 2011.

[129] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and Surviving Data Races Using Complementary Schedules. In *SOSP*, 2011.

[130] Christoph von Praun and Thomas Gross. Object Race Detection. In *OOPSLA*, 2001.

[131] Christoph von Praun and Thomas R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*, 2003.

[132] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *FSE*, 2007.

[133] Jaroslav Ševčík. Safe Optimisations for Shared-Memory Concurrent Programs. In *PLDI*, 2011.

[134] Edwin Westbrook, Jisheng Zhao, Zoran Budimli, and Vivek Sarkar. Practical Permissions for Race-Free Parallelism. In *ECOOP*, 2012.

[135] Benjamin Wester, David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Parallelizing Data Race Detection. In *ASPLOS*, 2013.

[136] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Data-Race Exceptions Have Benefits Beyond the Memory Model. In *MSPC*, 2011.

[137] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*, 2014.

[138] Benjamin P. Wood, Adrian Sampson, Luis Ceze, and Dan Grossman. Composable Specifications for Structured Shared-Memory Communication. In *OOPSLA*, 2010.

[139] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing Races in Live Applications with Execution Filters. In *OSDI*, 2010.

[140] Xinwei Xie and Jingling Xue. Acculock: Accurate and Efficient Detection of Data Races. In *CGO*, 2011.

[141] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*, 2005.

[142] Minjia Zhang, Jipeng Huang, , Man Cao, and Michael D. Bond. LarkTM: Efficient, Strongly Atomic Software Transactional Memory. Technical Report OSU-CISRC-11/12-TR17, Computer Science & Engineering, Ohio State University, 2012.

[143] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, 2007.