Low-Level Detection of Language-Level Data Races with LARD

Benjamin P. Wood Luis Ceze Dan Grossman
University of Washington
{bpw,luisceze,djg}@cs.washington.edu

Abstract

Researchers have proposed always-on *data-race exceptions* as a way to avoid the ill effects of data races, but slow performance of accurate dynamic data-race detection remains a barrier to the adoption of always-on data-race exceptions. Proposals for accurate low-level (*e.g.*, hardware) data-race detection have the potential to reduce this performance barrier. This paper explains why low-level data-race detectors are *wrong* for programs written in high-level languages (*e.g.*, Java): they miss true data races and report false data races in these programs. To bring the benefits of low-level data-race detection to high-level languages, we design *low-level abstractable race detection* (LARD), an extension of the interface between low-level data-race detectors and run-time systems that enables accurate language-level data-race detection using low-level detection mechanisms. We implement accurate LARD data-race exception support for Java, coupling a modified Jikes RVM Java virtual machine and a simulated hardware race detector. We evaluate our detector's accuracy against an accurate dynamic Java data-race detector and other low-level race detectors without LARD, showing that naïve accurate low-level data-race detectors suffer from many missed and false language-level races in practice, and that LARD prevents this inaccuracy.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—reliability; C.1.4 [*Processor Architectures*]: Parallel Architectures; D.3.4 [*Programming Languages*]: Processors—Run-time environments.

Keywords data-race detection; data-race exceptions; dynamic analysis; run-time systems

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/10.1145/2541940.2541955

1. Introduction

A *data race* is a pair of accesses to the same memory location by different threads where at least one access is a write and no synchronization orders the two accesses. Data races in shared-memory multithreaded programs notoriously result in problematic and confusing bugs. Data races also play a part in other multithreading errors such as atomicity violations or determinism violations and permit violations of sequential consistency due to reordering of memory operations by the compiler and hardware.

Some researchers have proposed *data-race exceptions* to mitigate the ill effects of data races by making *every* data race explicit at run-time [1, 10, 13, 24, 28]: on the second in a pair of accesses that race, raise an exception instead of doing the access. Data races become obvious at run-time like null pointer dereferences. Implementing *accurate* dynamic data-race detection, reporting a data race if and only if a true data race occurs, is challenging. Most techniques miss true races, report false races, or incur heavy performance overhead. We use *accurate* to mean sound *and* complete: an accurate dynamic data-race detector misses no true data races and reports no false data races in the execution it observes.

Recent proposals for low-level dynamic data-race detection have full accuracy and improved performance. RADISH [12] uses a mix of software and hardware support for data-race detection fast enough for many deployment situations. Aikido [34] 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 implementation allows for (1) hardware acceleration and (2) reuse of fast data-race detection mechanisms by many systems.

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 paper 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

race detection and high-level language implementations to remove all sources of missed races and false 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.

1.1 Low-Level Races ≠ Language-Level 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 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 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 race detector. Two broad features of the translation cause false and missed language-level data races for low-level 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 race detector can report false races. By analyzing distinct virtual memory locations when a single Java object has moved, the detector can miss true 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 races subject to thread placement.

1.2 Low-Level Detection of Language-Level Races

This paper's main contribution is *low-level abstractable race detection* (LARD), an extended low-level 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 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 paper is a simple execution stack consisting of a language run-time running on hardware, but the design may generalize to a range of dual-level execution environments, including operating systems, hypervisors, and interpreters.

To the best of our knowledge, this paper presents the first design for *virtualizing data-race detection*. Earlier exclusively low-level or language-level data-race detectors (*e.g.*, [12, 16, 44]) have support for marking custom synchronization routines to avoid false 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 race detectors (*e.g.*, [40, 44]) treat allocation specially to help reduce false races and some JVM race detectors work around object movement to avoid missed races (*e.g.*, [11, 37]), 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 paper, we characterize the effects of the language-level/low-level translation on race detection in depth and develop a unified approach to *eliminate all* missed and *all* false races, achieving accurate language-level data-race detection in high-level programs using low-level race detectors.

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 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 [12], and extended to support LARDx86. We modified Jikes RVM [2] to emit 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 races in practice.

1.4 Contributions and Outline

We discuss background on data races, data-race exceptions, and data-race detection in §2. Our main contributions follow:

- We explain why low-level race detectors are incorrect for high-level programs (§3), synthesizing disparate issues encountered in prior contexts [11, 37, 40, 46].

- We design *low-level abstractable race detection* (LARD), a simple interface for low-level 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 (§3).
- 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).
- We evaluate our implementation's accuracy, comparing against FastTrack [16], a naïve low-level data-race detector similar to RADISH [12], and various partial implementations of LARD, finding that, *in practice*, naïve ISA-level race detectors suffer from false and missed races for Java programs, but LARD does not (§5).

Finally, we discuss more related work (§6) and conclude (§7).

2. Background

2.1 Data Races and Data-Race Exceptions

A *data race* is defined formally as a pair of *concurrent*, *conflicting* memory accesses [32]. Two accesses *conflict* if they are executed by different threads and at least one access is a write. Operations are *concurrent* if they are not ordered by the *happens-before* relation [22], a partial order over operations, composed of *program order*, the order of operations within each thread, and *synchronization order*, the ordering between synchronization operations in different threads (*e.g.*, lock acquire and release or thread fork and join).

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*. The Java [26] and C/C++ [7] memory models both guarantee sequential consistency given data-race freedom, but if a data race occurs, a much weaker – or undefined – semantics applies [1, 6]. Even if sequential consistency is preserved [24, 28], data races play a part in other concurrency errors such as atomicity violations or determinism violations.

Data-Race Exceptions Some researchers have proposed *data-race exceptions* (*DREs*) to mitigate the ill effects of data races by making every data race explicit at run-time [1, 10, 13]. A memory model with accurate DREs 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 DREs. DREs ensure an execution forbids a data race before any ill effects can occur.

2.2 Dynamic Data-Race Detection

An accurate dynamic data-race detector misses no true data races and reports no false data races *in the execution it observes*. Races possible in other executions are irrelevant. We consider dynamic data-race detection with respect to the execution semantics and memory abstractions of the *source language*. A ***language-level data-race detector*** analyzes execution at the same level of abstraction provided by the source language. We typically mean a race detector for a language like Java, analyzing field accesses, etc. A ***low-level data-race detector*** analyzes execution at a machine abstraction below that of the source language. We typically mean hardware analyzing virtual memory accesses at the ISA level, but the same principles apply to a binary-instrumenting software implementation, for example.

The literature is flush with methods for dynamically detecting or avoiding data races; each is limited by accuracy or performance and designed for a specific programming language or execution abstraction. Some dynamic analyses allow false races or missed races in exchange for speed. Detection of data races that violate sequential consistency is fast [24, 28, 41], but does not detect *all* data races, which remain important. Atomicity violation detection, determinism enforcement, and other analyses often rely on accurate data-race detection [18] or assume data-race freedom [33]. Accurate software dynamic data-race detectors such as FastTrack [16, 17] and Goldilocks [13] miss no races and report no false races, but are too slow for always-on deployment, even after elimination of provably redundant checks [17].

2.3 Benefits of Low-Level Race Detection

Advances in accurate dynamic data-race detection have used low-level support to **improve performance** and accelerate checking in common cases where no data-sharing occurs. Mechanisms include page protections and hypervisor support in Aikido [34] or cache coherence tricks and specialized hardware in RADISH [12]. 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. Implementing the critical core of race detection logic once in a **general**, reusable, low-level mechanism is appealing. Although a low-level mechanism *alone* is insufficient for language-level race detection, it can *reduce* the complexity of high-performance race detection in a language implementation. Our implementation (§4) shows that the engineering required is feasible for a system as complex as a JVM. For unmanaged targets like C programs, it is simpler.

2.4 Vector-Clock Race Detectors

This paper considers data-race detectors derived from a canonical algorithm using vector clocks [14, 29] to track the happens-before order induced by synchronization and determine if pairs of conflicting memory accesses are concurrent. *Vector clocks* track the most recent operation in each thread that happens before a given event. A vector clock for each thread represents all operations that happen before that thread's next operation. A vector clock for each lock represents all operations that happen before that lock's last release.

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

Table 1. The LARD interface.

Thread and lock vector clocks are updated on synchronization. An *access history* for each shared memory location records the last write to the location by any thread and the last read from the location by each thread. When a thread accesses a location, it compares the access history with its own vector clock to check if earlier conflicting accesses happen before or race with its current access, then records its own access in the history. Writes must be totally ordered with respect to conflicting reads and writes, but reads are allowed to execute concurrently with other reads. We refer the reader to Appendix A for more detail.

3. 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 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 (§3.1), synchronization (§3.2), memory allocation (§3.3), memory movement (§3.4), and thread mapping (§3.5). For each operation, we show how information lost in translation can cause missed or false data races. Then we extend the low-level race detector interface to retain sufficient language-level information. Table 1 summarizes the five interface extensions. Finally, we argue that LARD's five extensions are sufficient to virtualize general low-level data-race detection for use by language implementations (§3.6).

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 §6.) However, we believe our research is the first to consider the full set of techniques needed to virtualize data-race detection.

3.1 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 race detector should check only accesses explicit in the source program for 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.

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 races that are bound to result. Figure 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 race detector will report a 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 race detector analyzes tracked accesses only.

3.2 Synchronization

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

RULE 2. *The race detector should analyze only synchronization explicit in the source program.*

Example Consider the upper left example in Figure 1, which shows views of the same multithreaded execution at the Java, JVM implementation, and low-level 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 race detector misses this data race if a

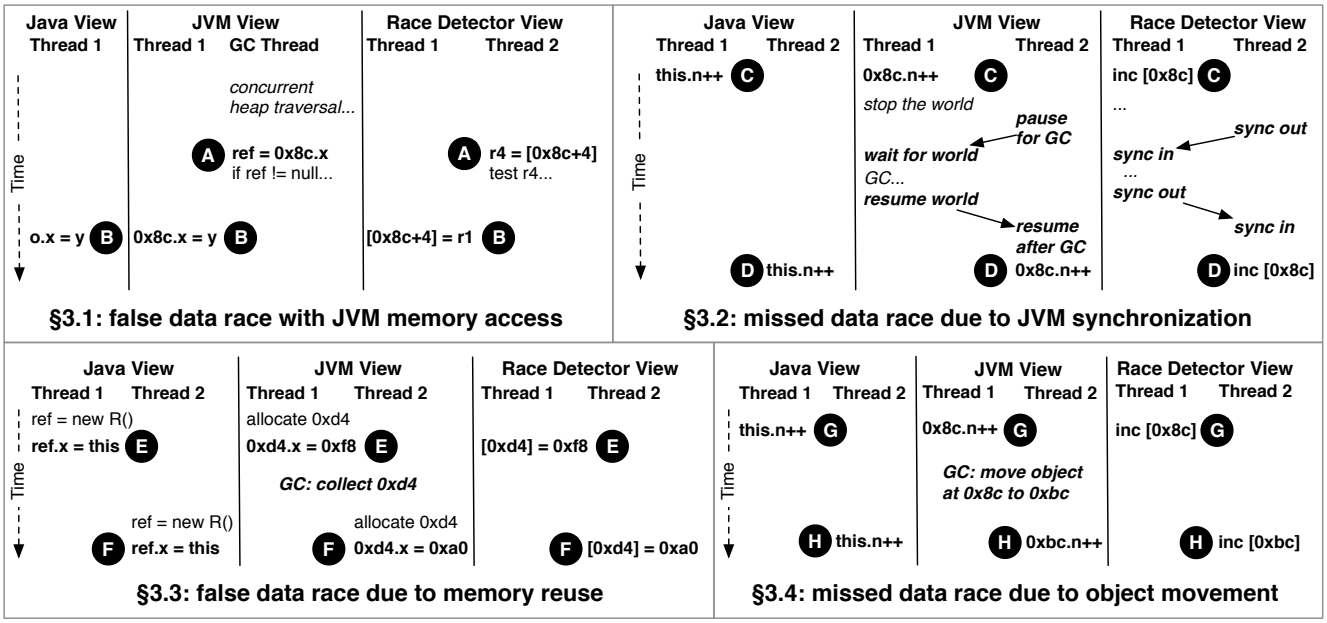

Figure 1. A naïve low-level data-race detector reports false data races and misses true data races in Java programs. Solid arrows are happens-before edges observed by the race detector. Each example contains three views of the same execution.

stop-the-world garbage collection occurs between the two accesses. If the 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 [12] provide primitives for synchronization libraries to report happens-before effects of synchronization to the race detector. This approach is easily adaptable for LARD: synchronization reports should be issued only for synchronization operations explicit in the source program.

3.3 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 race detector may report false races between accesses to the new and old language-level occupants.

RULE 3. *The 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 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 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 1. The 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 race detector, since it could also hide true data races in the Java program (§3.2). In the C/C++ memory model [9], 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 race detector access histories on deallocation. Thus newly allocated low-level memory locations appear fresh to the 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 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 [44] and RACEZ [40], two 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 race reports.

3.4 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 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 [11], §3.3. Similar issues arise for an operating system or hypervisor running above a physical-memory race detector when remapping virtual-memory pages.

RULE 4. *The race detector should move access histories of low-level memory locations to follow atomically with language-level memory objects moved by the run-time system.*

Example Consider the example execution in the lower right of Figure 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 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 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.*, [35]) 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 race detector in a way that ensures access history is moved atomically with program data. Neither the run-time system nor the race detector can accomplish this alone.

An alternative is to use logical addresses for race detection analysis so it is resilient to movement [37], 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 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 race detectors such as RADISH [12], and would require shootdown on object movement anyway. While reporting movement has overheads to copy access histories, race-checked memory accesses occur *far* more frequently than object movement. We choose reporting movement (copying) as a less invasive and higher-performance option.

3.5 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 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 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 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 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 race detector uses this thread identity for all operations of the low-level thread until the next such report.

3.6 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 §3.1-3.5 affect data-race detection. The five extensions described above ensure data-race detection is performed only on language-level memory and synchronization operations (§3.1, §3.2) and on the language's abstractions of memory (§3.3, §3.4) and thread identity (§3.5). 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-

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

race detection. This perspective was essential for guiding our implementation and evaluation.

LARD's effectiveness depends on the contract between the low-level race detector and the language implementation; it does *not* free the language implementation from reasoning about all details of 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 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 [26] and C/C++ [7] memory models may remove races from the original program, but none introduce races where races did not exist. *Roach-motel reordering* allows the movement of memory operations into – but not out of – critical sections [45]. 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 race. It is explicitly allowable behavior in the language memory model and, unlike with the issues above, the program can never manifest this race as compiled. If the language implementers *do* consider this a missed race, they must choose compiler optimizations appropriately. Regardless, this is the language implementer's choice and is an issue common to dynamic race detectors implemented at *all* levels of abstraction, not just low-level race detection for high-level languages.

4. 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 2): LARDx86 (§4.1) is an extension of the x86 ISA with LARD primitives. LARDISH (§4.2) is a simulated hardware implementation of LARDx86 that performs accurate, LARD-aware, data-race detection derived from the RADISH [12] hardware-based race detector. Jikes LARDVM (§4.3) 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 (§4.4).

4.1 The LARDx86 ISA

LARDx86 extends the x86 ISA to provide a LARD interface between software run-time systems and low-level vector-clock race detectors. To support the tasks described in §3, 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.

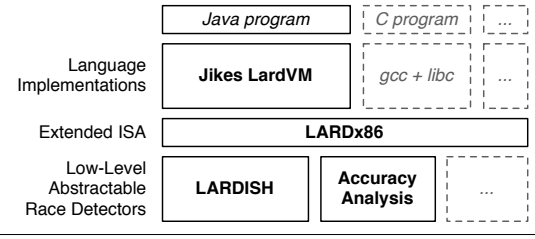

Figure 2. The LARD environment.

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 races by the low-level race detector. Untracked accesses are never checked. The low-level 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 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 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 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). 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 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 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 race detector to optimize bulk updates internally, but regions are restricted to a single virtual memory page to simplify support in hardware-based race detectors.

Encapsulated vs. Exposed Access Histories LARDx86 assumes that the low-level 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 race detector manages access histories based on physical – not virtual – addresses (§4.2). Reverse address translation or virtually addressed caches would be necessary to support efficient external software management of access histories.

4.2 The LARDISH Hardware Race Detector

To evaluate the performance potential of a hardware-based LARD system, we designed LARDISH, an extension of RADISH [12], a hardware-supported race detector. This section outlines our extensions to RADISH to support low-level abstractable race detection with LARDx86.

RADISH [12] is a hybrid hardware-software race detector that is accurate for low-level programs. It optimizes the canonical vector clock race detection algorithm by caching access histories in the hardware data cache, allowing most accesses to be checked for races purely in hardware. Several optimizations ensure that the common case race check has little or no latency. A software system handles storage of race detector access histories after cache evictions and context switches and support for occasional race checks that require access history information not cached in hardware. The software stores access histories in a format independent from the in-cache hardware representation.

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 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 a to 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 need 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 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 "underneath" them. 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 The Jikes LARDVM Java Virtual Machine

Jikes LARDVM is a modified version of Jikes RVM [2] 3.1.1 that implements accurate data-race detection for Java using LARDx86. Jikes LARDVM marks source program accesses for race checks (§4.3.1), tracks source program thread identity and synchronization (§4.3.2), and reports garbage collector operations (§4.3.3). Our implementation was designed for detailed accuracy analysis and is only lightly optimized for performance.

4.3.1 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 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.

4.3.2 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 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 [26], 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 Memory Management and Mapping

We modified the classical mark-sweep and semispace collectors in Jikes RVM and MMTk [4] 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 race-checked source program accesses and thus may have access histories in the race detector. Unlike synchronization and access, conservative reporting of memory-management events cannot cause missed or false 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.

4.3.4 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 (§4.4), extensive debugging/profiling, and significant code duplication for garbage collector access barriers.

The core LARD additions mainly track synchronization events 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 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.

4.4 Extensions for Accuracy Analysis

For the accuracy analysis in §5.2, we built an analysis tool that runs multiple 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 (§5.2) and not in real deployments.

5. 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 capacity to eliminate missed races and false races by comparing the results of our LARD-based Java data-race detector with the accurate FastTrack Java data-race detector [16] and a naïve low-level vector-clock data-race detector (§5.1) and quantify the effects of individual LARD extensions in terms of the missed races and false races they eliminate (§5.2). 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 (§5.3). Additionally, we evaluate the performance of Jikes LARDVM alone on conventional x86 hardware (§5.4).

Experimental Configuration We ran experiments with multithreaded benchmarks from Java Grande [43] and DaCapo 9.12 [3].[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 quad-core 64-bit 2.8GHz Intel Xeon Pentium 4 machines with 4GB of RAM and a Linux 2.6.32 kernel.

5.1 False Races and Missed Races

To validate the accuracy of our LARD implementation and illustrate the degree to which naïve low-level race detectors can suffer from false and missed races, we compare the race reports from three data-race detectors. **FastTrack** [16] is an accurate data-race detector for Java implemented via bytecode instrumentation. We compared FastTrack race reports with those of our **LARD** implementation (described in §4) and a **Naïve** low-level vector-clock race detector equivalent to RADISH [12]. 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.

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

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 2. Numbers of dynamic accesses on which data-race reports occur (DRE) and distinct PCs of these reports.

Table 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 races reported by each detector on any *single* execution. Column **DRE** shows the number of dynamic accesses on which 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 *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. 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 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 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 races reported. We carefully examined 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 races individually. These duplicated reports inflate LARD's race counts compared to those of FastTrack, though they report the same races, and account for the remaining disparities between LARD and FastTrack. We plan to remove this duplication in future implementations.

Comparison with a Naïve Low-Level Race Detector The Naïve configuration reports up to several orders of magnitude more races than LARD or FastTrack, as shown in Table 2. The majority of these are false races involving system accesses; some are false races between language-level accesses. The Naïve detector also misses some races reported by LARD or FastTrack, as discussed in §5.2, and reports races on a large number of PCs, including many outside the source program.

5.2 Impacts of LARD Extensions

To understand the practical impact of each LARD extension on missed and false 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 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 (§4.4).
- **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 (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 3 shows the results.

The LARD column repeats the count of racy accesses and distinct PCs for LARD from Table 2. For the other detectors, we report: (1) the number of false DREs (**F DRE**) – those accesses on which LARD does not report a 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 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 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 race reports on PCs in system code does not eliminate 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 races when memory reuse is ignored (**NoClear**), but avrora and sunflow do suffer false 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 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 races in data-race-free executions such as the Java Grande benchmarks, but missed races do occur under semispace collection in pmd and sunflow.

5.3 Jikes LARDVM Performance on LARDISH

We have begun 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 [12], extended with mechanisms described in §4.2. A PIN [25] binary instrumentation tool emulates LARDx86 to drive the simulator, an extension of the simulator used in [12]. 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

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 3. False/missed races with individual LARD extensions disabled.

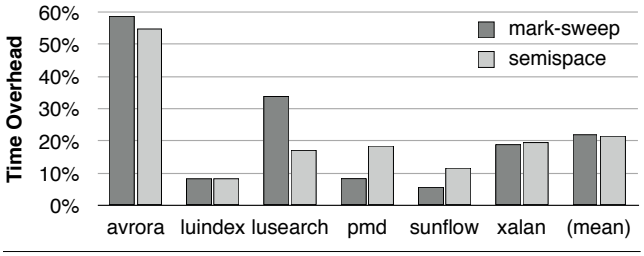

Figure 3. Execution time overhead of Jikes LARDVM normalized to unmodified Jikes RVM, both run on native x86.

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 [12], while lower on average. JVM accesses are not race-checked, so LARDISH can introduce less overhead than RADISH does for C programs in which all accesses are checked.

5.4 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 3 shows the additional overheads of Jikes LARDVM normalized to unmodified Jikes RVM using the same garbage 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.

6. Related Work

We believe LARD is the first approach for accurate dynamic data-race detection in high-level languages using hardware support or other low-level implementations.

6.1 General Data-Race Detection

Static data-race detectors, such as Chord [31] and RccJava [15], are imprecise, employing conservative analyses to avoid missing races at the cost of reporting false races. Race-free programming models such as Cyclone [19] and Deterministic Parallel Java [5] use conservative language restrictions to ensure race-freedom and other properties. Model checking approaches, as in Java RaceFinder [21], can use precise happens-before race detection to avoid reporting false races, but they remain subject to the limits of model checking, potentially missing races beyond the state-space they explore. This paper focuses on dynamic data-race detection, which can avoid precision problems for general programs.

This paper focuses on accurate dynamic data-race detection (§2). *Imprecise* dynamic data-race detectors use heuristics such as the *single protecting lock* pattern checked by the lockset algorithm in Eraser [38] to approximate data-race detection. While Eraser misses no races, it reports many false races since it does not reason about synchronization patterns beyond simple locking. Industrial-strength tools such as Helgrind [44], Google's ThreadSanitizer [39], and Intel's Inspector [20] also have limited precision and may impose high performance overheads. Other low-level data-race detectors have been proposed (CORD [36], HARD [48], SigRace [30],

etc.), but they do not guarantee accuracy. Other data-race detectors, such as LiteRace [27], Pacer [8], and SOS [23], start from accurate algorithms and use sampling or other methods to trade full accuracy for performance. While the inaccuracy of all of these techniques renders them unsuitable for our goal of accurate data-race exceptions, our approach nonetheless could be applied to fundamentally inaccurate detectors. LARD could also bring hardware support to dynamic data-race detectors that generalize the current execution to detect data races possible in other executions [42, 47].

6.2 Sequential Consistency Violation Exceptions

DRFx [28, 41] and Conflict Exceptions [24] are proposals for precise low-level detection of data races that may violate sequential consistency. While LARD addresses full data-race detection, some of the same translation problems arise when using low-level support to check language-level sequential consistency. For example, ensuring sequential consistency for Java may not require sequential consistency of the full low-level execution, if run-time systems use careful lock-free techniques. Regardless, the stronger property of data-race freedom remains desirable in many contexts. Full data-race detection is a part of many other concurrency analyses.

6.3 Memory Management and Synchronization Issues

Others have previously identified memory management or custom synchronization as potential sources of inaccuracy, but have provided only partial solutions in general.

Helgrind [44] offers C preprocessor macros to annotate custom synchronization and allocation in C/C++ programs to reduce false race reports, but there is no way to turn off analysis of synchronization instrumented by default (*e.g.*, pthreads) or to specify memory movement in a way that preserves data-race detection access history. LARD provides a general interface to communicate the salient details of language-level synchronization and memory abstractions to the low-level race detector for accurate data-race detection.

Choi, *et al.*, mention that memory reuse and movement in garbage collection could break their JVM-level race detector, since it stores addresses of objects in its analysis state [11]. Their solution, a large enough heap size that garbage collection never occurs, is not feasible for production systems. Qi, *et al.*, address this problem in their MulticoreSDK [37] JVM race detector implementation with logical addresses for race-checked locations. As discussed in §3.4, this approach adds overhead to the memory access critical path and is poorly suited for low-level race detectors. LARD uses explicit movement of analysis state to follow the movement of objects, which occurs much less frequently than memory accesses.

RACEZ [40] is an offline, dynamic data-race detector for C/C++ that uses the imprecise lockset algorithm [38], sampling of memory accesses via hardware performance monitoring, and offline log analysis to improve run-time performance at the cost of missing some true races and reporting some false 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.

7. 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 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.

Acknowledgments

This work was supported in part by NSF grants CCF-1016495 and CCF-0811405; and C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Several colleagues contributed insights to this project: Joe Devietti assisted with the RADISH simulator; Laura Effinger-Dean discussed the C/C++ memory model semantics for allocation; Sebastian Burckhardt confirmed that thread identity can be an issue in analyses at the kernel level; Hadi Esmailzadeh, Jacob Nelson, and Adrian Sampson provided insights on hardware design; Tom Bergan, Joe Devietti, and Brandon Lucia gave helpful feedback on the manuscript. Finally, our anonymous reviewers gave insightful and constructive feedback that improved the presentation of our work.

References

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

[2] 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.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, 2006.

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

[5] 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.

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

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

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] Colin Fidge. Logical Time in Distributed Computing Systems. *Computer*, 24, August 1991.

[15] Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000.

[16] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[17] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant Check Elimination For Dynamic Race Detectors. In *ECOOP*, 2013.

[18] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A Sound And Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, 2008.

[19] Dan Grossman. Type-Safe Multithreading in Cyclone. In *TLDI*, 2003.

[20] Intel. Inpector XE. http://software.intel.com/en-us/intel-inspector-xe, 2013.

[21] Huafeng Jin, Tuba Yavuz-Kahveci, and Beverly A. Sanders. Java Memory Model-Aware Model Checking. *TACAS*, 7214, 2012.

[22] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21, July 1978.

[23] Du Li, Witawas Srisa-an, and Matthew B. Dwyer. SOS: Saving Time in Dynamic Race Detection with Stationary Analysis. In *OOPSLA*, 2011.

[24] 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.

[25] 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.

[26] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL*, 2005.

[27] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-race Detection. In *PLDI*, 2009.

[28] 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.

[29] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[30] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, 2009.

[31] Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. In *PLDI*, 2006.

[32] 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.

[33] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[34] Marek Olszewski, Qin Zhao, David Koh, Jason Ansel, and Saman Amarasinghe. Aikido: Accelerating Shared Data Dynamic Analyses. In *ASPLOS*, 2012.

[35] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A Study of Concurrent Real-Time Garbage Collectors. In *PLDI*, 2008.

[36] Milos Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In *HPCA*, 2006.

[37] 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.

[38] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *TOCS*, 15(4), 1997.

[39] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[40] 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.

[41] 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.

[42] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound Predictive Race Detection in Polynomial Time. In *POPL*, 2012.

[43] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *Supercomputing*, 2001.

[44] Valgrind Project. Helgrind: a thread error detector. http://valgrind.org/docs/manual/hg-manual.html, 2013.

[45] Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.

[46] Benjamin P. Wood, Luis Ceze, and Dan Grossman. Data-Race Exceptions Have Benefits Beyond the Memory Model. In *MSPC*, 2011.

[47] Xinwei Xie and Jingling Xue. Acculock: Accurate and Efficient Detection of Data Races. In *CGO*, 2011.

[48] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, 2007.

A. A Canonical Race Detector

The algorithm optimized by the state-of-the-art accurate software [16] and hardware [12] race detectors uses *vector clocks* [14, 29] to represent the happens-before graph. Each thread maintains its own local integer logical time, starting at zero. A *vector clock* contains an integer logical clock for each thread, indexed by thread ID. Each thread maintains a vector clock representing the last logical time in each thread that happened before the current logical time in this thread. (The ith entry in thread i's vector clock is its local logical time.) Each lock in the program also has an associated vector clock representing the last logical time in each thread that happened before the last release of the lock.

Synchronization updates the vector clocks of threads and locks as follows: 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. When thread t joins thread u, thread t's vector clock is updated to be the pairwise maximum of the current vector clocks of threads t and u. 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. When thread t releases lock l, lock l's vector clock is updated to the pairwise 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.) Other types of synchronization are handled similarly: incoming synchronization merges into the thread's vector clock and outgoing synchronization merges from the thread's vector clock and then advances the thread's local time.

Each memory location has an access history storing the thread that performed the last write to the location and the local time at which it performed that write, as well as, for each thread, the last local time at which that thread read from the location. When a memory access is executed, it is first checked against the access history. On any access to x, we check that the current thread's vector clock entry for the thread t that last wrote x is greater than or equal to the time at which t last wrote to x. On a write to x, we check, for each thread t, that the current thread's vector clock entry for t is greater than or equal to the time at which t last read from x. If any of these checks fails, a data race is reported, otherwise the access is safe and it must be recorded in the access history. For a write, we set the last writer of x to the current thread and the last write clock to be the current thread's current local time. As an optimization, we typically zero all the last reads as well. On a write, we set the last read of x by the current thread to the current thread's current local time.