Valor: Efficient, Software-Only Region Conflict Exceptions *

Swarnendu Biswas Minjia Zhang Michael D. Bond
Ohio State University (USA)
{biswass,zhanminj,mikebond}@cse.ohio-state.edu

Brandon Lucia
Carnegie Mellon University (USA)
blucia@cmu.edu

Abstract

Data races complicate programming language semantics, and a data race is often a bug. Existing techniques detect data races and define their semantics by detecting conflicts between *synchronization-free regions* (*SFRs*). However, such techniques either modify hardware or slow programs dramatically, preventing always-on use today.

This paper describes *Valor*, a sound, precise, software-only region conflict detection analysis that achieves high performance by eliminating the costly analysis on each read operation that prior approaches require. Valor instead logs a region's reads and *lazily* detects conflicts for logged reads when the region ends. As a comparison, we have also developed *FastRCD*, a conflict detector that leverages the epoch optimization strategy of the FastTrack data race detector.

We evaluate Valor, FastRCD, and FastTrack, showing that Valor dramatically outperforms FastRCD and FastTrack. Valor is the first region conflict detector to provide strong semantic guarantees for racy program executions with under 2X slowdown. Overall, Valor advances the state of the art in always-on support for strong behavioral guarantees for data races.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—monitors, testing tools; D.3.4 [*Programming Languages*]: Processors—compilers, run-time environments

Keywords Conflict exceptions; data races; dynamic analysis; region serializability

* This material is based upon work supported by the National Science Foundation under Grants CSR-1218695, CAREER-1253703, and CCF-1421612.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3689-5/15/10...$15.00
http://dx.doi.org/10.1145/2814270.2814292

1. Introduction

Data races are a fundamental barrier to providing well-defined programming language specifications and to writing correct shared-memory, multithreaded programs. A *data race* occurs when two accesses to the same memory location are *conflicting*—executed by different threads and at least one is a write—and *concurrent*—not ordered by synchronization operations [4].

Data races can cause programs to exhibit confusing and incorrect behavior. They can lead to sequential consistency (SC), atomicity, and order violations [42, 54] that may corrupt data, cause a crash, or prevent forward progress [28, 36, 52]. The Northeastern electricity blackout of 2003 [66], which was caused by a data race, is a testament to the danger posed by data races.

The complexity and risk associated with data races motivate this work. The thesis of our work is that systems should furnish programming language implementations with a mechanism that gives data races clear, simple semantics. However, building a system that *efficiently* provides strong semantic guarantees for data races is complex and challenging—and has remained elusive. As a case in point: Java and C++ support variants of *DRF0*, which provides essentially no useful guarantees about the semantics of data races [2, 3, 10, 11, 44, 70] (Section 2). These languages remain useful, nonetheless, by providing clear, intuitive semantics *for programs that are free of data races*. For data-race-free programs, DRF0 guarantees not only SC but also serializability of synchronization-free regions (SFRs) [3, 12, 44, 45, 54]. Our goal in this work is to develop a mechanism that equips present and future languages with similarly clear, intuitive semantics *even for programs that permit data races*.

One way to deal with data races is to detect *problematic* data races and halt the execution with an exception when one occurs [24, 44, 47]. Problematic data races are ones that may violate a useful semantic property, like sequential consistency [24] or SFR serializability [44]. Exceptional semantics for data races have several benefits [44]. First, they simplify programming language specifications because data races have clear semantics, and an execution is either correct or it throws an exception. Second, they make software safer by limiting the possible effects of data races. Third, they permit aggres-

sive optimizations within SFRs that might introduce incorrect behavior with weaker data race semantics. Fourth, they help debug problematic data races by making them a fail-stop condition. Treating problematic data races as exceptions requires a mechanism that dynamically detects those races precisely (no false positives) and is efficient enough for always-on use. *In this work, we develop a mechanism that meets this requirement by detecting problematic races efficiently and precisely.*

Existing approaches. Existing sound[1] and precise dynamic data race detectors slow program executions by an order of magnitude or more in order to determine which conflicting accesses are concurrent according to the happens-before relation [24, 27, 38, 55] (Section 2). Other prior techniques avoid detecting happens-before races soundly, instead detecting conflicts only between operations in concurrent SFRs [23, 44, 47]. Every SFR conflict corresponds to a data race, but not every data race corresponds to an SFR conflict. Detecting SFR conflicts provides the useful property that an execution with no region conflicts corresponds to a serialization of SFRs. This guarantee extends to *executions with data races* the strong property that DRF0 already provides for data-race-free executions [2, 3, 12, 45]. Unfortunately, existing region conflict detectors are impractical, relying on custom hardware or slowing programs substantially [23, 44, 47].

Our Approach

This work aims to provide a practical, efficient, software-only region conflict detector that is useful for giving data races clear semantics. Our key insight is that *tracking the last readers of each shared variable is not necessary for sound and precise region conflict detection*. As a result of this insight, we introduce a novel sound and precise region conflict detection analysis called *Valor*. Valor records the last region to *write* each shared variable, but it does *not* record the last region(s) to *read* each variable. Instead, it *logs* information about each read in thread-local logs, so that each thread can later *validate* its logged reads to ensure that no conflicting writes occurred in the meantime. Valor thus detects write–write and write–read conflicts *eagerly* (i.e., at the second conflicting access), but it detects read–write conflicts *lazily*.

We note that some software transactional memory (STM) systems make use of similar insights about eager and lazy conflict detection, although STM mechanisms need not be precise (Section 7.3). Our work shows how to apply insights about mixed eager–lazy conflict detection to precise region conflict detection.

To better understand the characteristics and performance of Valor, we have also developed *FastRCD*, which is an adaptation of the efficient *FastTrack* happens-before data race detector [27] for region conflict detection. FastRCD does not track the happens-before relation; instead it tracks regions that last wrote and read each shared variable. As such, FastRCD provides somewhat lower overhead than FastTrack but still incurs most of FastTrack's costs by tracking the last reader(s) to each shared variable.

We have implemented FastRCD and Valor, as well as the state-of-the-art FastTrack analysis [27], in a Java virtual machine (JVM) that has performance competitive with commercial JVMs (Appendix E). We evaluate and compare the performance, runtime characteristics, data race detection coverage, scalability, and space overheads of these three analyses on a variety of large, multithreaded Java benchmarks. Valor incurs the lowest overheads of any sound, precise, software conflict detection system that we are aware of, adding only 99% average overhead.

By contrast, our implementation of FastTrack adds 342% average overhead over baseline execution, which is comparable to 750% overhead reported by prior work [27]—particularly in light of implementation differences (Section 6.2 and Appendix D). FastRCD incurs most but not all of FastTrack's costs, adding 267% overhead on average.

Valor is not only substantially faster than existing approaches that provide strong semantic guarantees,[2] but its <2X average slowdown is fast enough for pervasive use during development and testing, including end-user alpha and beta tests, and potentially in some production systems. Valor thus represents a significant advancement of the state of the art: the first approach with under 100% time overhead on commodity systems that provides useful, strong semantic guarantees to existing and future languages for executions both with and without data races.

2. Background and Motivation

Assuming system support for sound, precise data race detection enables a language specification to clearly and simply define the semantics of data races, but sound and precise dynamic data race detection adds high run-time overhead [27]. An alternative is to detect conflicts between synchronization-free regions (SFRs), which also provides guarantees about the behavior of executions with data races, but existing approaches rely on custom hardware [44, 47, 65] or add high overhead [23].

Providing a strong execution model. Modern programming languages including Java and C++ have memory models that are variants of the *data-race-free-0* (DRF0) memory model [3, 12, 45], ensuring data-race-free (DRF) executions are *sequentially consistent* [39] (SC). As a result, DRF0 also provides the stronger guarantee that DRF executions correspond to a serialization of SFRs [2, 3, 44].

Unfortunately, DRF0 provides no semantics for executions with data races; the behavior of a C++ program with a

[1] In this paper, a dynamic analysis is *sound* if it never incurs false negatives for the analyzed execution.

[2] In fairness, FastTrack's guarantees and goals differ from Valor and FastRCD's: unlike Valor and FastRCD, FastTrack detects every data race in an execution.

data race is undefined [12]. A recent study emphasizes the difficulty of reasoning about data races, showing that a C/C++ program with seemingly "benign" data races may behave incorrectly due to compiler transformations or architectural changes [10]. Java attempts to preserve memory and type safety for executions with data races by avoiding "out-of-thin-air" values [45], but researchers have shown that it is difficult to prohibit such values without precluding common compiler optimizations [13, 70].

These deficiencies of DRF0 create an urgent need for systems to clearly and simply define the semantics of executions with data races and provide strong guarantees about their behavior [2, 11, 18]. Recent work gives *fail-stop semantics* to data races, treating a data race as an *exception* [18, 24, 44, 47, 65]. Our work is motivated by these efforts, and our techniques also give data races fail-stop semantics.

Detecting data races soundly and precisely. Sound and precise dynamic data race detection enables a language to define the semantics of data races by throwing an exception for every data race. To detect races soundly and precisely, an analysis must track the *happens-before* relation [38]. Analyses typically track happens-before using vector clocks; each vector clock operation takes time proportional to the number of threads. In addition to tracking happens-before, an analysis must track *when* each thread last wrote and read each shared variable, in order to check that each access *happens after* every earlier conflicting access. *FastTrack* reduces the cost of tracking last accesses, without missing any data races, by tracking a single last writer and, in many cases, a single last reader [27].

Despite this optimization, FastTrack still slows executions by nearly an order of magnitude on average [27]. Its high run-time overhead is largely due to the cost of tracking shared variable accesses, especially *reads*. A program's threads may perform reads concurrently, but FastTrack requires each thread to update shared metadata on each read. These updates effectively convert the reads into writes that cause remote cache misses. Moreover, FastTrack must synchronize to ensure that its happens-before checks and metadata updates happen atomically. These per-read costs fundamentally limit FastTrack and related analyses.

Detecting region conflicts. Given the high cost of sound, precise happens-before data race detection, prior work has sought to detect the subset of data races that may violate serializability of an execution's SFRs—SFR serializability being the same guarantee provided by DRF0 for DRF executions. Several techniques detect conflicts between operations in SFRs that overlap in time [23, 44, 47]. SFR conflict detection yields the following guarantees: a DRF execution produces no conflicts; any conflict is a data race; and a conflict-free execution is a serialization of SFRs.

Prior work on *Conflict Exceptions* detects conflicts between overlapping SFRs [44] and treats conflicts as exceptions; these guarantees are essentially the same as those provided by our work. Conflict Exceptions achieves high performance via hardware support for conflict detection that augments existing cache coherence mechanisms. However, its hardware support has several drawbacks. First, it adds complexity to the cache coherence mechanism. Second, each cache line incurs a high space overhead for storing metadata. Third, sending larger coherence messages that include metadata leads to coherence network congestion and requires more bandwidth. Fourth, cache evictions and synchronization operations for regions with evictions become more expensive because of the need to preserve metadata by moving it to and from memory. Fifth, requiring new hardware precludes use in today's systems.

DRFx detects conflicts between regions that are synchronization free but also *bounded*, i.e., every region has a bounded maximum number of instructions that it may execute [47]. Bounded regions allow DRFx to use simpler hardware than Conflict Exceptions [47, 65], but DRFx cannot detect all violations of SFR serializability, although it guarantees SC for conflict-free executions. Like Conflict Exceptions, DRFx is inapplicable to today's systems because it requires hardware changes.

IFRit detects data races by detecting conflicts between overlapping *interference-free regions* (*IFRs*) [23]. An IFR is a region of one thread's execution that is associated with a particular variable, during which another thread's read and/or write to that variable is a data race. IFRit relies on whole-program static analysis to place IFR boundaries conservatively, so IFRit is precise (i.e., no false positives). Conservatism in placing boundaries at data-dependent branches, external functions calls, and other points causes IFRit to miss some IFR conflicts. In contrast to our work, IFRit does not aim to provide execution model guarantees, instead focusing on precisely detecting as many races as possible.

The next section introduces our analyses that are entirely software based (like IFRit) and detect conflicts between full SFRs (like Conflict Exceptions).

3. Efficient Region Conflict Detection

The goal of this work is to develop a region conflict detection mechanism that is useful for providing guarantees to a programming language implementation, and is efficient enough for always-on use. (We defer discussing what defines a region's boundaries until Section 3.3.1. Briefly, our analyses can use regions demarcated by all synchronization operations, or by synchronization "release" operations only.)

We explore two different approaches for detecting region conflicts. The first approach is *FastRCD*, which, like FastTrack [27], uses *epoch optimizations* and *eagerly* detects conflicts at conflicting accesses. We have developed FastRCD in order to better understand the characteristics and performance of a region conflict detector based on FastTrack's approach. Despite FastRCD being a natural extension

of the fastest known sound and precise dynamic data race detection analysis, Section 6.2 experimentally shows that FastRCD's need to track last readers imposes overheads that are similar to FastTrack's.

In response to FastRCD's high overhead, we develop ***Valor***,[3] *which is the main contribution of this work*. Valor detects write–write and write–read conflicts *eagerly* as in FastRCD. The key to Valor is that it detects read–write conflicts *lazily*, effectively avoiding the high cost of tracking last-reader information. In Valor, each thread logs read operations locally. At the end of a region, the thread *validates* its read log, checking for read–write conflicts between its reads and any writes in other threads' ongoing regions. By *lazily* checking for these conflicts, Valor can provide fail-stop semantics without hardware support and with overheads far lower than even our optimized FastRCD implementation.

Section 3.1 describes the details of FastRCD and the fundamental sources of high overhead that eager conflict detection imposes. Section 3.2 then describes Valor and the implications of lazy conflict detection. Sections 3.3 and 4 describe extensions and optimizations for FastRCD and Valor.

3.1 FastRCD: Detecting Conflicts Eagerly in Software

This section presents FastRCD, a new software-only dynamic analysis for detecting region conflicts. FastRCD reports a conflict when a memory access executed by one thread conflicts with a memory access that was executed by another thread in a region that is ongoing. It provides essentially the same semantics as Conflict Exceptions [44] but without hardware support.

In FastRCD, each thread keeps track of a clock c that starts at 0 and is incremented at every region boundary. This clock is analogous to the logical clocks maintained by FastTrack to track the happens-before relation [27, 38].

FastRCD uses *epoch optimizations* based on FastTrack's optimizations [27] for efficiently tracking read and write metadata. It keeps track of the single last region to write each shared variable, and the last region or regions to read each shared variable. For each shared variable x, FastRCD maintains x's last writer region using an epoch $\mathsf{c@t}$: the thread t and clock c that last wrote to x. When x has no concurrent reads from overlapping regions, FastRCD represents the last reader as an epoch $\mathsf{c@t}$. Otherwise, FastRCD keeps track of last readers in the form of a *read map* that maps threads to the clock values c of their last read to x. We use the following notations to help with exposition:

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

epoch(T) – Returns an epoch $\mathsf{c@T}$, where c represents the ongoing region in thread T.

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

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

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

Algorithms 1 and 2 show FastRCD's analysis at program writes and reads, respectively. At a write by thread T to program variable x, the analysis first checks if the last writer epoch matches the current epoch, indicating an earlier write in the same region, in which case the analysis does nothing (line 1). Otherwise, it checks for conflicts with the previous write (lines 3–4) and reads (lines 5–7). Finally, it updates the metadata to reflect the current write (lines 8–9).

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

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

At a read, the instrumentation first checks for an earlier read in the same region, in which case the analysis does nothing (line 1). Otherwise, it checks for a conflict with a prior write by checking if the last writer thread t is still executing its region c (lines 3–4). Finally, the instrumentation updates T's clock in the read map (line 5).

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

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

FastRCD's analysis at a read or write must execute *atomically*. Whenever the analysis needs to update x's metadata ($\mathcal{W}_x$ and/or $\mathcal{R}_x$), it "locks" x's metadata for the duration of the action (not shown in the algorithms). Because the analyses presented in Algorithms 1 and 2 read and write multiple metadata words, the analyses are not amenable to a "lock-free" approach that updates the metadata using a single atomic operation.[4] Note that the analysis and program memory access

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

[4] Recent Intel processors provide Transactional Synchronization Extensions (TSX) instructions, which support multi-word atomic operations via hardware transactional memory [73]. However, recent work shows that existing TSX implementations incur high per-transaction costs [48, 60].

need *not* execute together atomically because the analysis need not detect the order in which conflicting accesses occur, just that they conflict.

FastRCD soundly and precisely detects every region conflict just before the conflicting access executes. FastRCD guarantees that region-conflict-free executions are region serializable, and that every region conflict is a data race. It suffers from high overheads (Section 6.2) because it unavoidably performs expensive analysis at reads. Multiple threads' concurrent regions commonly read the same shared variable; updating per-variable metadata at program reads leads to communication and synchronization costs *not* incurred by the original program execution.

3.2 Valor: Detecting Read–Write Conflicts Lazily

This section describes the design of *Valor*, a novel, software-only region conflict detector that eliminates the costly analysis on read operations that afflicts FastRCD (and FastTrack). Like FastRCD, Valor reports a conflict when a memory access executed by one thread conflicts with a memory access previously executed by another thread in a region that is ongoing. Valor soundly and precisely detects conflicts that correspond to data races and provides the same semantic guarantees as FastRCD. Valor detects write–read and write–write conflicts exactly as in FastRCD, but detects read–write conflicts differently. Each thread locally logs its current region's reads and detects read–write conflicts *lazily* when the region ends. Valor eliminates the need to track the last reader of each shared variable explicitly, avoiding high overhead.

3.2.1 Overview

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

This section presents Valor so that it tracks each shared variable's last writer using an epoch, just as for FastRCD. Section 4 presents an *alternate* design of Valor that represents the last writer differently using *ownership* information (for implementation reasons). Conceptually, the two designs work in the same way, e.g., the examples in this section apply to the design in Section 4.

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

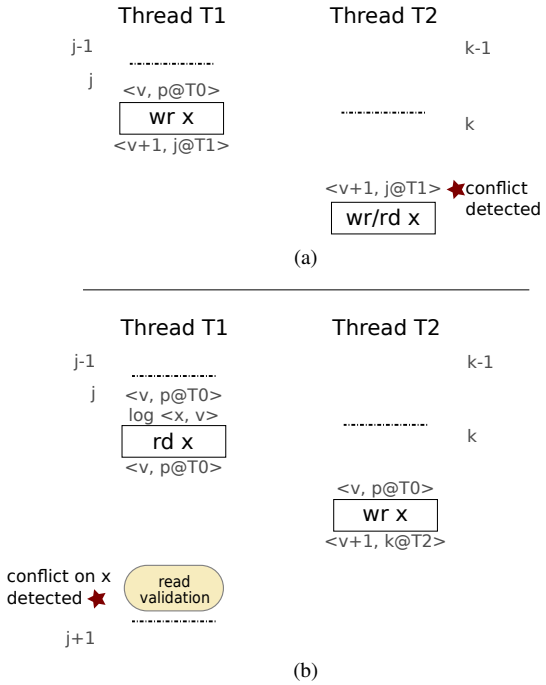


Figure 1. (a) Like FastRCD, Valor eagerly detects a conflict at T2's access because the last region to write x is ongoing. (b) *Unlike* FastRCD, Valor detects read–write conflicts lazily. During read validation, T1 detects a write to x since T1's read of x.

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

Read–write conflicts. Figure 1(b) shows an example read–write conflict. At the read of x, T1 records the read in its thread-local *read log*. A read log entry, ⟨x, v⟩, consists of the address of variable x and x's current version, v.

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

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

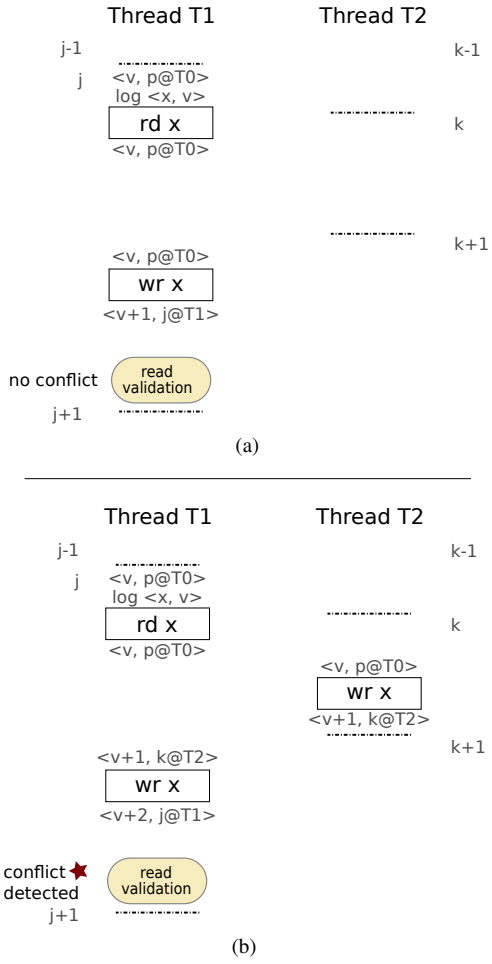


Figure 2. Valor relies on versions to detect conflicts soundly and precisely.

conflicting write, which Section 3.2.3 argues is acceptable for providing strong behavior guarantees.

Valor requires versions. Tracking epochs alone is insufficient: Valor's metadata must include versions. Let us assume for exposition's sake that Valor tracked only epochs and not versions, and it recorded epochs instead of versions in read logs, e.g., $\langle \mathsf{x}, \mathsf{p@T0} \rangle$ in Figure 1(b). In this particular case, Valor without versions correctly detects the read–write conflict in Figure 1(b).

However, in the general case, Valor without versions is either unsound or imprecise. Figures 2(a) and 2(b) illustrate why epochs alone are insufficient. In Figure 2(a), no conflict exists. The analysis should *not* report a conflict during read validation, because even though x's epoch has changed from the value recorded in the read log, T1 itself is the last writer.

In Figure 2(b), T1 is again the last writer of x, but in this case, T1 *should* report a read–write conflict because of T2's intervening write. (No *write–write* conflict exists: T2's region k ends before T1's write.) However, using epochs alone, Valor cannot differentiate these two cases during read validation.

Thus, Valor uses versions to differentiate cases like Figures 2(a) and 2(b). Read validation detects a conflict for x if (1) its version has changed and its last writer thread is not the current thread **or** (2) its version has changed at least *twice*,[5] definitely indicating intervening write(s) by other thread(s).

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

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

3.2.2 Analysis Details

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

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

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

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

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

As in Section 3.1, we use T for the current thread, and t and t' for other threads.

Analysis at writes. Algorithm 3 shows the analysis that Valor performs at a write. It does nothing if x's last writer epoch matches the current thread T's current epoch (line 2), indicating that T has already written to x. Otherwise, the analysis checks for a write–write conflict (lines 3–4) by checking if c = clock(t), indicating that x was last written by an ongoing region in another thread (note that this situation implies t ≠ T). Finally, the analysis updates $\mathcal{W}_x$ with an incremented version and the current thread's epoch (line 5).

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

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

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

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

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

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

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

Analysis at region end. Valor detects read–write conflicts lazily at region boundaries, as shown in Algorithm 5. For each entry $\langle \mathsf{x}, \mathsf{v} \rangle$ in the read log, the analysis compares v with x's current version v'. Differing versions are a necessary but insufficient condition for a conflict. If x was last written by the thread ending the region, then a difference of *more* than one (i.e., $\mathsf{v}' \geq \mathsf{v}+2$) is necessary for a conflict (line 3).

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

```
1: for all ⟨x, v⟩ ∈ T.readLog do
2:     let ⟨v', c@t⟩ ← W_x
3:     if (v' ≠ v ∧ t ≠ T) ∨ v' ≥ v + 2 then
4:         Conflict!                ▷ Read–write conflict detected
5: T.readLog ← ∅
```

We note that when Valor detects a write–write or write–read conflict, it is not necessarily the *first* conflict to occur: there may be an earlier read–write conflict waiting to be detected lazily. To report such read–write conflicts first, Valor triggers read validation before reporting a detected write–write or write–read conflict.

Atomicity of analysis operations. Similar to FastTrack and FastRCD, Valor's analysis at writes, reads, and region boundaries must execute atomically in order to avoid missing conflicts and corrupting analysis metadata. Unlike FastTrack and FastRCD, Valor can use a lock-free approach because the analysis accesses a single piece of shared state, $\mathcal{W}_x$. The write analysis updates $\mathcal{W}_x$ (line 5 in Algorithm 3) using an atomic operation (not shown). If the atomic operation fails because another thread updates $\mathcal{W}_x$ concurrently, the write analysis restarts from line 1. At reads and at region end, the analysis does not update shared state, so it does not need atomic operations.

3.2.3 Providing Valor's Guarantees

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

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

Deferred detection does *not* compromise Valor's semantic guarantees as long as the effects of conflicting regions do not become externally visible. A region that performs a read that conflicts with a later write can behave in a way that would be *impossible in any unserializable execution*. We refer to such regions as "zombie" regions, borrowing terminology from software transactional memory (STM) systems that experience similar issues by detecting conflicts lazily [34]. To prevent external visibility, Valor must validate a region's reads before all sensitive operations, such as system calls and I/O. Similarly, a zombie region might never end (e.g., might get stuck in an infinite loop), even if such behavior is impossible under any region serializable execution. To account for this possibility, Valor must periodically validate reads in a long-running region. Other conflict and data race detectors have detected conflicts asynchronously [21, 47, 65], providing imprecise exceptions and similar guarantees.

In an implementation for a memory- and type-unsafe language such as C or C++, a zombie region could perform arbitrary behavior such as corrupting arbitrary memory. This issue is problematic for lazy STMs that target C/C++, since a transaction can corrupt memory arbitrarily, making it impossible to preserve serializability [20]. The situation is not so dire for Valor, which detects region conflicts in order to throw conflict exceptions, rather than to preserve region serializability. As long as a zombie region does not actually corrupt *Valor's analysis state*, read validation will be able to detect the conflict when it eventually executes—either when the region ends, at a system call, or periodically (in case of an infinite loop).

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

3.3 Extending the Region Conflict Detectors

This section describes extensions that apply to both FastRCD and Valor.

3.3.1 Demarcating Regions

The descriptions of FastRCD and Valor so far do not define the exact boundaries of regions. *Synchronization-free regions* (SFRs) are one possible definition that treats each synchronization operation as a region boundary [44, 47]. We observe that it is also correct to bound regions only at synchronization *release* operations (e.g., lock release, monitor wait, and thread fork) because region conflicts are still guaranteed to be true data races. We call these regions *release-free regions* (RFRs).

Figure 3 illustrates the difference between SFRs and RFRs. We note that the boundaries of SFRs and RFRs are determined dynamically (at run time) by the synchronization operations that execute, as opposed to being determined statically at

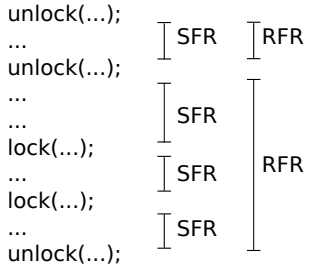


Figure 3. Synchronization- and release-free regions.

compile time. An RFR is at least as large as an SFR, so an RFR conflict detector detects at least as many conflicts as an SFR conflict detector. Larger regions potentially reduce fixed per-region costs, particularly the cost of updating writer metadata on the first write in each region.

There are useful analogies between RFR conflict detection and prior work. Happens-before data race detectors increment their epochs at release operations only [27, 55], and some prior work extends redundant instrumentation analysis past acquire, but not release, operations [30].

Correctness of RFR conflict detection. Recall that Valor and FastRCD define a region conflict as an access executed by one thread that conflicts with an already-executed access by another thread in an ongoing region. This definition is in contrast to an *overlap-based* region conflict definition that reports a conflict whenever two regions that contain conflicting accesses overlap at all. Both of these definitions support conflict detection between SFRs with no false positives. However, only the definition that we use for Valor and FastRCD supports conflict detection between RFRs without false data races; an overlap-based definition of RFR conflicts would yield false races. Appendix B proves the absence of false data races for our RFR conflict detection scheme.

3.3.2 Reporting Conflicting Sites

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

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

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

4. Alternate Metadata and Analysis for Valor

As presented in the last section, Valor maintains an epoch $c@t$ (as well as a version v) for each variable x. An epoch enables a thread to query whether an ongoing region has written x. An alternate way to support that query is to track *ownership*: for each variable x, maintain the thread t, if any, that has an ongoing region that has written x.

This section proposes an alternate design for Valor that tracks ownership instead of using epochs. For clarity, the rest of this paper refers to the design of Valor described in Section 3.2 as *Valor-E* (Epoch) and the alternate design introduced here as *Valor-O* (Ownership).

We implement and evaluate *Valor-O* for implementation-specific reasons: (1) our implementation targets IA-32 (Section 5); (2) metadata accesses must be atomic (Section 3.2.2); and (3) Valor-O enables storing metadata (ownership and version) in 32 bits.

We do not expect either design to perform better in general. Valor-O consumes less space and uses slightly simpler conflict checks, but it incurs extra costs to maintain ownership: in order to clear each written variable's ownership at region end, each thread must maintain a "write set" of variables written by its current region.

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

Analysis at writes. Algorithm 6 shows Valor-O's analysis at program writes. If T is already x's owner, it can skip the rest of the analysis since the current region has already written x (line 2). Otherwise, if x is owned by a concurrent thread, it indicates a region conflict (lines 3–4). T updates x's write metadata to indicate ownership by T and to increment the version number (line 5).

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

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

A thread relinquishes ownership of a variable only at the next region boundary. To keep track of all variables owned by a thread's region, each thread T maintains a *write set*, denoted by T.writeSet (line 6), which contains all shared variables written by T's current region.

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

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

1: **let** $\langle \mathsf{v}, \mathsf{t} \rangle \leftarrow \mathcal{W}_x$
2: **if** $\mathsf{t} \neq \phi \wedge \mathsf{t} \neq \mathsf{T}$ **then**
3: Conflict! ▷ Write–read conflict detected
4: $\mathsf{T}.\mathsf{readLog} \leftarrow \mathsf{T}.\mathsf{readLog} \cup \{\langle \mathsf{x}, \mathsf{v} \rangle\}$

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

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

1: **for all** $\langle \mathsf{x}, \mathsf{v} \rangle \in \mathsf{T}.\mathsf{readLog}$ **do**
2: **let** $\langle \mathsf{v}', \mathsf{t} \rangle \leftarrow \mathcal{W}_x$
3: **if** $(\mathsf{v}' \neq \mathsf{v} \wedge \mathsf{t} \neq \mathsf{T}) \vee \mathsf{v}' \geq \mathsf{v} + 2$ **then**
4: Conflict! ▷ Read–write conflict detected
5: $\mathsf{T}.\mathsf{readLog} \leftarrow \emptyset$
6: **for all** $\mathsf{x} \in \mathsf{T}.\mathsf{writeSet}$ **do**
7: **let** $\langle \mathsf{v}, \mathsf{t} \rangle \leftarrow \mathcal{W}_x$ ▷ Can assert $\mathsf{t} = \mathsf{T}$
8: $\mathcal{W}_x \leftarrow \langle \mathsf{v}, \phi \rangle$ ▷ Remove ownership by T
9: $\mathsf{T}.\mathsf{writeSet} \leftarrow \emptyset$

Finally, the analysis at region end processes the write set by setting the ownership of each owned variable to ϕ (lines 6–8) and then clearing the write set (line 9).

5. Implementation

We have implemented FastTrack, FastRCD, and Valor in Jikes RVM 3.1.3 [6],[6] a Java virtual machine that performs competitively with commercial JVMs (Appendix E). Our implementations share features as much as possible: they instrument the same accesses, and FastRCD and Valor demarcate regions in the same way. As Section 4 mentioned, we implement the *Valor-O* design of Valor.

We have made our implementations publicly available.[7]

5.1 Features Common to All Implementations

The implementations target IA-32 and extend Jikes RVM's *baseline* and *optimizing* dynamic compilers, to instrument synchronization operations and memory accesses. The implementations instrument all code in the *application context*, including application code and library code (e.g., `java.*`) called from application code.[8]

Instrumenting program operations. Each implementation instruments synchronization operations to track happens-before (FastTrack) or to demarcate regions (FastRCD and Valor). Acquire operations are lock acquire, monitor resume, thread start and join, and volatile read. Release operations are lock release, monitor wait, thread fork and terminate, and volatile write. By default, FastRCD and Valor detect conflicts between release-free regions (RFRs; Section 3.3.1) and add no instrumentation at acquires.

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

Eliminating redundant instrumentation. We have implemented an intraprocedural dataflow analysis to identify *redundant* instrumentation points. Instrumentation on an access to x is redundant if it is definitely preceded by an access to x in the same region (cf. [15, 30]). Specifically, instrumentation at a write is redundant if preceded by a write, and instrumentation at a read is redundant if preceded by a read or write. The implementations eliminate redundant instrumentation by default, which we find reduces *the run-time overheads added* by FastTrack, FastRCD, and Valor by 3%, 4%, and 5%, respectively (results not shown).

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

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

5.2 FastTrack and FastRCD

The FastRCD implementation shares many features with our FastTrack implementation, which is faithful to prior work's implementation [27]. Both implementations increment a thread's logical clock at each synchronization release operation, and they track last accesses similarly. Both maintain each shared variable's last writer and last reader(s) using FastTrack's epoch optimizations. In FastTrack, if the prior read is an epoch that *happens before* the current read, the algorithm continues using an epoch, and if not, it upgrades to a read map. FastRCD uses a read epoch if the last reader region has ended, and if not, it upgrades to a read map. Each read map is an efficient, specialized hash table that maps threads to clocks. We modify garbage collection (GC) to check each variable's read metadata and, if it references a read map, to trace the read map.

We represent FastTrack's epochs with two (32-bit) words. We use 9 bits for thread identifiers, and 1 bit to differentiate a read epoch from a read map. Encoding the per-thread clock

[6] `http://www.jikesrvm.org`

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

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

with 22 bits to fit the epoch in one word would cause the clock to overflow, requiring a separate word for the clock.

FastRCD represents epochs using a single 32-bit word. FastRCD avoids overflow by leveraging the fact that it is always correct to reset all clocks to either 0, which represents a completed region, or 1, which represents an ongoing region. To accommodate this strategy, we modify GC in two ways. (1) Each full-heap GC sets every variable's clock to 1 if it was accessed in an ongoing region and to 0 otherwise. (2) Each full-heap GC resets each thread's clock to 1. Note that FastTrack cannot use this optimization.

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

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

Tracking happens-before. In addition to instrumenting acquire and release synchronization operations as described in Section 5.1, FastTrack tracks the happens-before edge from each static field initialization in a class initializer to corresponding uses of that static field [41]. The FastTrack implementation instruments static (including `final`) field loads as an acquire of the same lock used for class initialization, in order to track those happens-before edges.

5.3 Valor

We implement the Valor-O design of Valor described in Section 4.

Tracking the last writer. Valor tracks the last writer in a single 32-bit metadata per variable: 23 bits for the version and 9 bits for the thread. Versions are unlikely to overflow because variables' versions are independent, unlike overflow-prone clocks, which are updated at every region boundary. We find that versions overflow in only two of our evaluated programs. A version overflow could lead to a missed conflict (i.e., a false negative) if the overflowed version happened to match some logged version. To mitigate version overflow, Valor could reset versions at full-heap GCs, as FastRCD resets its clocks (Section 5.2).

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

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

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

6. Evaluation

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

6.1 Methodology

Benchmarks. We evaluate our implementations using large, realistic, benchmarked applications: the DaCapo benchmarks [8] with the *large* workload size, versions 2006-10-MR2 and 9.12-bach (distinguished with names suffixed by 6 and 9); and fixed-workload versions of SPECjbb2000 and SPECjbb2005.[9] We omit single-threaded programs and programs that Jikes RVM 3.1.3 cannot execute.

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

We limit the GC to 4 threads instead of the default 64 because of a known scalability bottleneck in Jikes RVM's memory management toolkit (MMTk) [22]. Using 4 GC threads improves performance for all configurations and the baseline. This change leads to reporting *higher* overheads for FastTrack, FastRCD, and Valor than with 64 GC threads, since less time is spent in GC, so the time added for conflict detection is a greater fraction of baseline execution time.

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

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

6.2 Run-Time Overhead

Figure 4 shows the overhead added over unmodified Jikes RVM by the different implementations. Each bar is the average of 10 trials and has a 95% confidence interval that is centered at the mean. *The main performance result in this paper is that Valor incurs only 99% run-time overhead on*

[9] `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

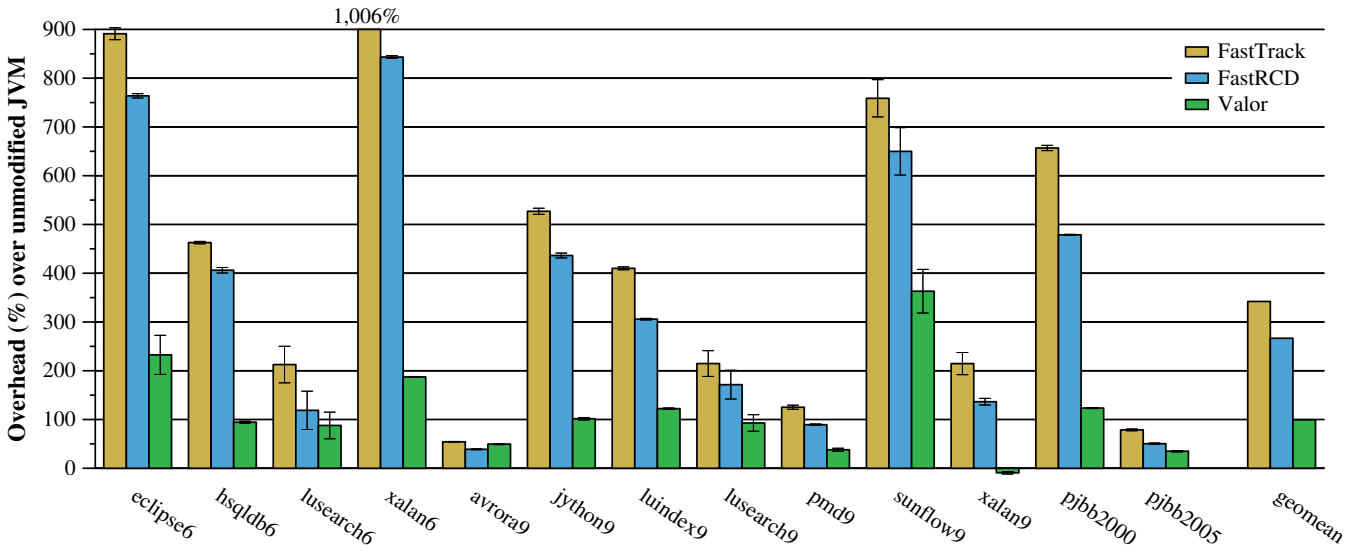


Figure 4. Run-time overhead added to unmodified Jikes RVM by our implementations of FastTrack, FastRCD, and Valor.

average, far exceeding the performance of any prior conflict detection technique. We discuss Valor's performance result in context by comparing it to FastTrack and FastRCD.

FastTrack. Our FastTrack implementation adds 342% overhead on average (i.e., 4.4X slowdown). Prior work reports an 8.5X average slowdown, but for a different implementation and evaluation [27]. In Appendix D, we compare the two FastTrack implementations in our evaluation setting.

FastRCD. Figure 4 shows that FastRCD adds 267% overhead on average. FastRCD tracks accesses similarly to FastTrack, but has lower overhead than FastTrack because it does not track happens-before. We measured that around 70% of FastRCD's cost comes from tracking last readers; the remainder comes from tracking last writers, demarcating regions, and bloating objects with per-variable metadata. Observing the high cost of last reader tracking motivates Valor's lazy read validation mechanism.

Valor. Valor adds only 99% overhead on average, which is substantially lower than the overheads of any prior software-only technique, including our FastTrack and FastRCD implementations. The most important reason for this improvement is that Valor completely does away with expensive updates and synchronization on last reader metadata. Valor consistently outperforms FastTrack and FastRCD for all programs except avrora9, for which FastTrack and FastRCD add particularly low overhead (for unknown reasons). Valor slightly outperforms the baseline for xalan9; we believe this unintuitive behavior is a side effect of reactive Linux thread scheduling decisions, as others have observed [7].

6.3 Scalability

This section evaluates how the run-time overhead of Valor, compared with FastRCD and FastTrack, varies with additional application threads—an important property as systems increasingly provide more cores. We use the three evaluated programs that support spawning a configurable number of application threads: lusearch9, sunflow9, and xalan9 (Table 1). Figure 5 shows the overhead for each program over the unmodified JVM for 1–64 application threads, using the configurations from Figure 4. Figure 5 shows that all three techniques' overheads scale with increasing numbers of threads. Valor in particular provides similar or decreasing overhead as the number of threads increases.

6.4 Space Overhead

This section evaluates the space overhead added by FastTrack, FastRCD, and Valor. We measure an execution's space usage as the maximum memory used after any full-heap garbage collection (GC). Our experiments use Jikes RVM configured with the default, high-performance, generational GC and let the GC adjust the heap size automatically (Section 6.1).

Figure 6 shows the space overhead, relative to baseline (unmodified JVM) execution for the same configurations as in Figure 4. We omit luindex9 since the unmodified JVM triggers no full-heap GCs, although each of the three analyses does. FastTrack, FastRCD, and Valor add 180%, 112%, and 98%, respectively. Unsurprisingly, FastTrack uses more space than FastRCD since it maintains more metadata. Valor sometimes adds less space than FastRCD; other times it adds more. This result is due to the analyses' different approaches for maintaining read information: FastRCD uses per-variable shared metadata, whereas Valor logs reads in per-thread buffers. On average, Valor uses less memory than FastRCD and a little more than half as much memory as FastTrack.

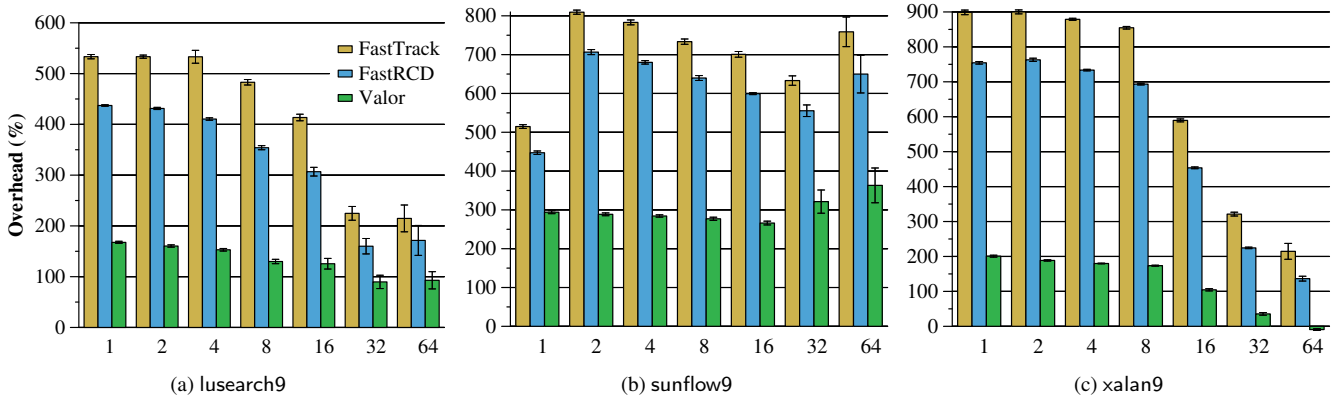


Figure 5. Run-time overheads of the configurations from Figure 4, for 1–64 application threads. The legend applies to all graphs.

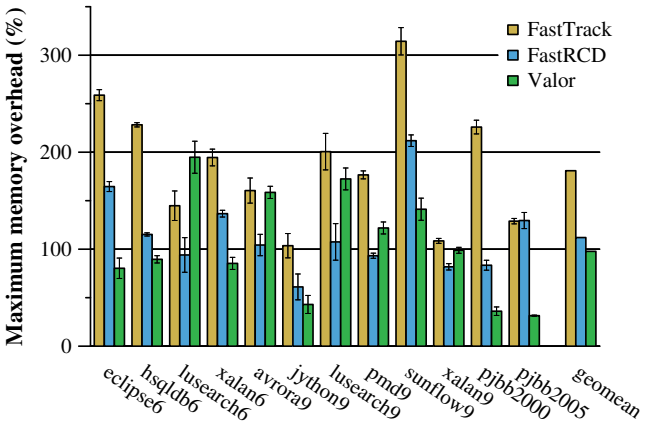


Figure 6. Space overheads of the configurations from Figure 4.

6.5 Run-Time Characteristics

Table 1 characterizes the evaluated programs' behavior. Each value is the mean of 10 trials of a statistics-gathering version of one of the implementations. The first two columns report the total threads created and the maximum active threads at any time.

The next columns, labeled *Reads* and *Writes*, report instrumented, executed read and write operations (in millions). The *No metadata updates* columns show the percentage of instrumented accesses for which instrumentation need not update or synchronize on any metadata. For FastTrack, these are its "read/write same epoch" and "read shared same epoch" cases [27]. For FastRCD and Valor, these are the cases where the analysis does not update any per-variable metadata. Note that Valor has no *Reads* column because it does not update per-variable metadata on a program read.

For three programs, FastTrack and FastRCD differ significantly in how many reads require metadata updates (minor differences for other programs are not statistically significant). These differences occur because the analyses differ in when they upgrade from a read epoch to a read map (Section 5.2). For per-*write* metadata updates, the analyses report very similar percentages, so we report a single percentage (the percentage reported by FastTrack).

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

6.6 Data Race Detection Coverage

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

Table 2 shows how many data races each analysis detects. A data race is defined as an unordered pair of static program locations. If the same race is detected multiple times in an execution, we count it only once. The first number for each detector is the average number of races (rounded to the nearest whole number) reported across 10 trials. Run-to-run variation is typically small: 95% confidence intervals are consistently smaller than $\pm 10\%$ of the reported mean, except for xalan9, which varies by $\pm 35\%$ of the mean. The number in parentheses is the count of races reported at least once across all 10 trials.

As expected, FastTrack reports more data races than FastRCD and Valor. On average across the programs, one run of either FastRCD or Valor detects 58% of the true data races. Counting data races reported at least once across 10 trials, the percentage increases to 63% for FastRCD and 73% for Valor, respectively. Compared to FastTrack, FastRCD and Valor represent lower coverage, higher performance points

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

Table 1. Run-time characteristics of the evaluated programs, executed by implementations of FastTrack, FastRCD, and Valor. Counts are rounded to three significant figures and the nearest whole number. Percentages are rounded to the nearest 0.1%. *Three programs by default spawn threads in proportion to the number of cores (64 in most of our experiments).

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

Table 2. Data races reported by FastTrack, FastRCD, and Valor. For each analysis, the first number is average distinct races reported across 10 trials. The second number (in parentheses) is distinct races reported at least once over all trials.

in the performance–coverage tradeoff space. We note that FastRCD and Valor are *able* to detect any data race, because any data race can manifest as a region conflict [23].

We emphasize that although FastRCD and Valor miss some data races, the reported races involve accesses that are dynamically "close enough" together to jeopardize region serializability (Section 2). We (and others [23, 44, 47]) argue that region conflicts are therefore *more* harmful than other data races, and it is more important to fix them.

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

Comparing SFR and RFR conflict detection. FastRCD and Valor bound regions at releases only, potentially detecting more races at lower cost as a result. We have evaluated the benefits of using RFRs in Valor by comparing with a version of Valor that uses SFRs. For every evaluated program, there is no statistically significant difference in races detected between SFR- and RFR-based conflict detection (10 trials each; 95% confidence). RFR-based conflict detection does, however, outperform SFR-based conflict detection, adding 99% versus 104% overhead on average, respectively. This difference is due to RFRs being larger and thus incurring fewer metadata and write set updates (Section 3.3.1).

6.7 Summary

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

7. Related Work

Section 2 covered the closest related work [23, 24, 27, 44, 47]. This section compares our work to other approaches.

7.1 Detecting and Eliminating Data Races

Software-based dynamic analysis. Happens-before analysis soundly and precisely detects an execution's data races, but it slows programs by about an order of magnitude (Section 2) [27]. An alternative is *lockset* analysis, which de-

tects violations of a locking discipline (e.g., [19, 62, 67]). However, lockset analysis reports false data races, so it is unsuitable for providing a strong execution model. Hybrids of happens-before and lockset analysis tend to report false positives (e.g., [53]).

Goldilocks [24] detects races soundly and precisely and provides exceptional, fail-stop data race semantics. The Goldilocks paper reports 2X average slowdowns, but the authors of FastTrack argue that a realistic implementation would incur an estimated 25X average slowdown [27].

In concurrent work, *Clean* detects write–write and write–read data races but does not detect read–write races [63]. Like Valor, Clean exploits the insight that detecting read–write conflicts eagerly is expensive. Unlike Valor, Clean does not detect read–write conflicts at all, but instead argues that detecting write–write and write–read data races is sufficient to provide certain execution guarantees such as freedom from out-of-thin-air values [45].

Other race detection approaches give up soundness entirely, missing data races in exchange for performance, usually in order to target production systems. *Sampling* and *crowdsourcing* approaches trade coverage for performance by instrumenting only some accesses [14, 26, 37, 46]. These approaches incur the costs of tracking the happens-before relation [14, 37, 46] and/or provide limited coverage guarantees [26, 46]. Since they miss data races, they are unsuitable for providing a strong execution model.

Hardware support. Custom hardware can accelerate data race detection by adding on-chip memory for tracking vector clocks and extending cache coherence to identify shared accesses [4, 21, 49, 63, 72, 74]. However, manufacturers have been reluctant to change already-complex cache and memory subsystems substantially to support race detection.

Static analysis. Whole-program static analysis considers all possible program behaviors (e.g., inputs, environments, and thread interleavings) and thus can avoid false negatives [25, 50, 51, 56, 69]. However, static analysis abstracts data and control flow conservatively, leading to imprecision and false positives. Furthermore, its imprecision and performance tend to scale poorly with increasing program size and complexity.

Leveraging static analysis. Whole-program static analysis can soundly identify definitely data-race-free accesses, which dynamic race detectors need not instrument. Prior work that takes this approach can reduce the cost of dynamic analysis somewhat but not enough to make it practical for always-on use [19, 24, 40, 68]. These techniques typically use static analyses such as thread escape analysis and thread fork–join analysis. Whole-program static analysis is not well suited for dynamically loaded languages such as Java, since all of the code may not be available in advance.

Our FastTrack, FastRCD, and Valor implementations currently employ intraprocedural static redundancy analysis to identify accesses that do not need instrumentation (Section 5.1). These implementations could potentially benefit from more powerful static analyses, although practical considerations (e.g., dynamic class loading and reflection) and inherent high imprecision for large, complex applications, limit the real-world opportunity for using static analysis to optimize dynamic analysis substantially.

Languages and types. New languages can eliminate data races, but they require writing programs in these potentially restrictive languages [9, 59]. Type systems can ensure data race freedom, but they typically require adding annotations and modifying code [1, 16].

Exposing effects of data races. Prior work exposes erroneous behavior due to data races, often under non-SC memory models [17, 28, 36, 52]. We note that every data race is potentially harmful because DRF0-based memory models provide no or very weak semantics for data races [2, 11, 52].

7.2 Enforcing Region Serializability

An alternative to detecting region conflicts is to *enforce* end-to-end region serializability. Existing approaches either enforce serializability of full synchronization-free regions (SFRs) [54] or bounded regions [5, 64]. They rely on support for expensive speculation that often requires complex hardware support.

Other dynamic approaches can tolerate the effects of data races by providing isolation from them [57, 58], but the guarantees are limited.

7.3 Detecting Conflicts

Software transactional memory (STM) detects conflicts between programmer-specified regions [33, 34]. To avoid the cost of tracking each variable's last readers, many STMs use so-called "invisible readers" and detect read–write conflicts *lazily* [34]. In particular, *McRT-STM* and *Bartok-STM* detect write–write and write–read conflicts eagerly and read–write conflicts lazily [35, 61]. These STMs validate reads differently from Valor: if a thread detects a version mismatch for an object that it last wrote, it detects a write by an intervening transaction either by looking up the version in a write log [61], or by waiting to update versions until a transaction ends (which requires read validation to check each object's ownership) [35].

In contrast, Valor avoids the costs of maintaining this data by checking if the version has increased by at least 2. Another difference is that Valor must detect conflicts precisely, whereas STMs do not (a false conflict triggers an unnecessary abort and retry). As a result, STMs typically track conflicts at the granularity of objects or cache lines. More generally, STMs have not introduced designs that target region conflict detection or precise exceptions. In some sense, our work applies insights from STMs to the context of data race exceptions.

RaceTM uses *hardware* TM to detect conflicts that are data races [32]. RaceTM is thus closest to existing hardware-based conflict detection mechanisms [44, 47, 65].

Last writer slicing (LWS) tracks data provenance, recording only the last writers of data [43]. LWS and our work share the intuition that to achieve low run-time overheads, a dynamic analysis should track only the last writer of each shared memory location. LWS is considerably different from our work in its purpose, focusing on understanding concurrency bugs by directly exposing last writer information in a debugger. LWS cannot detect read–write conflicts, and it does not detect races or provide execution model guarantees.

8. Conclusion

This work introduces two new software-based region conflict detectors, one of which, Valor, has overheads low enough to provide practical semantic guarantees to a language specification. The key insight behind Valor is that detecting read–write conflicts lazily retains necessary semantic guarantees and has better performance than eager conflict detection. Overall, Valor represents an advance in the state of the art for providing strong guarantees for racy executions. This advance helps make it practical to use all-the-time conflict exceptions in various settings, from in-house testing to alpha and beta testing to even some production systems.

Acknowledgments

We thank Steve Freund for helpful feedback on FastTrack experiments and on the text. We thank Man Cao, Joe Devietti, Jake Roemer, Michael Scott, and Aritra Sengupta for helpful discussions and advice. We thank the anonymous reviewers for detailed and insightful feedback on the text.

References

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

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

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

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

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

[6] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

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

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[32] S. Gupta, F. Sultan, S. Cadambi, F. Ivančić, and M. Rötteler. Using Hardware Transactional Memory for Data Race Detection. In *IPDPS*, pages 1–11, 2009.

[33] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, pages 388–402, 2003.

[34] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[35] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *PLDI*, pages 14–25, 2006.

[36] B. Kasikci, C. Zamfir, and G. Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, pages 185–198, 2012.

[37] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced Data Race Detection. In *SOSP*, pages 406–422, 2013.

[38] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

[39] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Computer*, 28:690–691, 1979.

[40] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *PLDI*, pages 463–474, 2012.

[41] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 2nd edition, 1999.

[42] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[43] B. Lucia and L. Ceze. Data Provenance Tracking for Concurrent Programs. In *CGO*, pages 146–156, 2015.

[44] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[45] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[46] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, pages 134–143, 2009.

[47] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[48] H. S. Matar, I. Kuru, S. Tasiran, and R. Dementiev. Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support. In *WoDet*, 2014.

[49] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, pages 337–348, 2009.

[50] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *POPL*, pages 327–338, 2007.

[51] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI*, pages 308–319, 2006.

[52] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*, pages 22–31, 2007.

[53] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *PPoPP*, pages 167–178, 2003.

[54] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. ...and region serializability for all. In *HotPar*, 2013.

[55] E. Pozniansky and A. Schuster. MultiRace: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *CCPE*, 19(3):327–340, 2007.

[56] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, pages 320–331, 2006.

[57] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs. In *ASPLOS*, pages 181–192, 2009.

[58] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and Tolerating Asymmetric Races. In *PPoPP*, pages 173–184, 2009.

[59] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *TOPLAS*, 20:483–545, 1998.

[60] C. G. Ritson and F. R. Barnes. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *CPA*, pages 271–292, 2013.

[61] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.

[62] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, pages 27–37, 1997.

[63] C. Segulja and T. S. Abdelrahman. Clean: A Race Detector with Cleaner Semantics. In *ISCA*, pages 401–413, 2015.

[64] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, pages 561–575, 2015.

[65] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, pages 53–66, 2011.

[66] U.S.–Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada. Technical report, Department of Energy, 2004.

[67] C. von Praun and T. R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

[68] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*, pages 115–128, 2003.

[69] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/FSE*, pages 205–214, 2007.

[70] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, pages 27–51, 2008.

[71] J. Wilcox, P. Finch, C. Flanagan, and S. N. Freund. Array Shadow State Compression for Precise Dynamic Race Detection. Technical Report CSTR-201510, Williams College, 2015.

[72] B. P. Wood, L. Ceze, and D. Grossman. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*, pages 671–686, 2014.

[73] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*, pages 19:1–19:11, 2013.

[74] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, pages 121–132, 2007.

A. Valor Is Sound and Precise

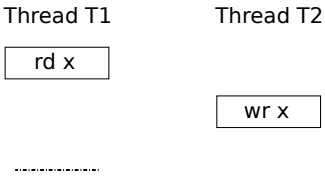
This section proves that Valor detects region conflicts soundly and precisely.[10] That is, it reports a conflict if and only if an execution has a region conflict, which is defined as an access that conflicts with an access executed in an ongoing region (Section 3). Here we assume that the relatively straightforward FastRCD algorithm detects region conflicts soundly and precisely.

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

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

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

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



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

At T1's read to x, let v be x's version (from $\mathcal{W}_x$). T1 logs $\langle x, v\rangle$ at the read (Algorithms 4 and 7). When T1's region ends, it performs read validation, which checks the following condition for the read log entry $\langle x, v\rangle$ (Algorithms 5 and 8):

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

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

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

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

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

Case 1: $v' = v$
: Since Valor increments versions monotonically, $v' = v$ only if T2's write does not increment x's version, which happens only if T2's region has already written x (Algorithms 3 and 6). We assumed that T2's write to x is the first write since T1's read, so T2's region must have written x *before* T1's read. By definition of region conflict, a write–read region conflict exists, which Valor detects, contradicting the initial assumption.

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

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

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

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

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

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

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

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

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

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

Case 1: $v' \neq v \ \wedge \ t \neq T$
: Because $v' \neq v$ (and only Valor's write analysis updates $\mathcal{W}_x$), there must have been a write by t to x between T's read and the region end that updated $\mathcal{W}_x$ to $\langle v', c@t\rangle$ (Valor-E's representation; Algorithm 3) or $\langle v', t\rangle$ (Valor-O's representation; Algorithm 6). Based on our initial assumption of region conflict freedom, this write must have been executed by T. Thus, $t = T$, contradicting this case's premise.

Case 2: $v' \geq v + 2$
: In order to increment x's version at least twice between T's read and region end, at least two writes in distinct regions must have written x (Algorithms 3 and 6). Only one of these writes can be in T's read's region, so the other write must be by a different thread, which by definition is a read–write region conflict, which contradicts the initial assumption.

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

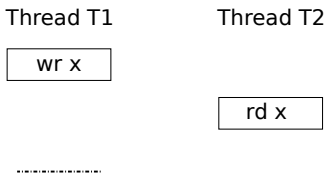
B. RFR Conflicts Are Data Races

This section proves the following theorem from Section 3.3.1, which applies to all of our region conflict detectors (FastRCD, Valor-E, and Valor-O):

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

Proof. We prove this claim by contradiction. Suppose that an RFR conflict exists in a data-race-free (DRF) execution. Recall that, by definition, an RFR conflict exists when an

access conflicts with another access executed by an ongoing RFR. Without loss of generality, we assume that a read by thread T2 conflicts with a write in an ongoing RFR in T1:



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

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

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

where $\prec_{HB}$ is the *happens-before* relation, a partial order that is the union of *program* order (i.e., intra-thread order) $\prec_{PO}$ and *synchronization* order $\prec_{SO}$ [38, 45].

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

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

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

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

C. Architectural Sensitivity

This section evaluates the sensitivity of our experiments to the CPU architecture by repeating our performance experiments on an Intel Xeon E5-4620 system with four 8-core processors (32 cores total). Otherwise, the methodology is the same as in Section 6.2. Figure 7 shows the overhead added over unmodified Jikes RVM by our implementations. FastTrack adds an overhead of 408%, while FastRCD adds 303% overhead. Valor continues to substantially outperform the other techniques, adding an overhead of only 116%.

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

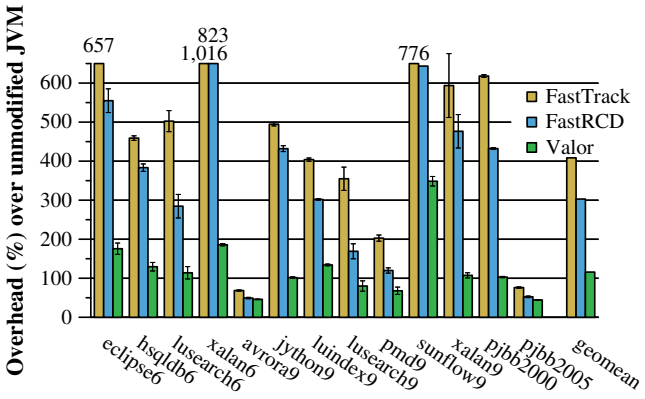


Figure 7. Run-time overhead added to unmodified Jikes RVM by our implementations of FastTrack, FastRCD, and Valor on an Intel Xeon E5-4620 system. Other than the platform, the methodology is the same as for Figure 4.

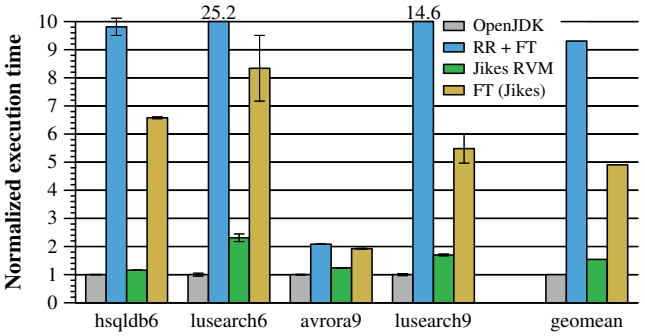


Figure 8. Performance comparison of FastTrack implementations. The last two configurations correspond to the baseline and FastTrack configurations in Figure 4.

D. Comparing FastTrack Implementations

To fairly and directly compare FastTrack, FastRCD, and Valor, we have implemented all three approaches in Jikes RVM (Section 5). This section seeks to better understand the performance differences between our FastTrack implementation and Flanagan and Freund's publicly available FastTrack implementation [27].[11] Their FastTrack implementation is built on the *RoadRunner* dynamic bytecode instrumentation framework, which alone slows programs by 4–5X on average [27, 29]. We execute the RoadRunner FastTrack implementation on a different JVM, Open JDK 1.7, because Jikes RVM would not execute it correctly. RoadRunner does not fully support instrumenting the Java libraries (e.g., java.*), and it does not support the class loading pattern used by the DaCapo harness [31], so we are only able to execute a few programs successfully, and we exclude library instrumentation. (Recent work runs the DaCapo benchmarks successfully

[11] https://github.com/stephenfreund/RoadRunner

with RoadRunner by running the programs after extracting them from the harness [71].)

Figure 8 shows how the implementations compare for the programs that RoadRunner executes. For each program, the first two configurations execute with OpenJDK, and the last two execute with Jikes RVM. The results are normalized to the first configuration, which is unmodified OpenJDK. The second configuration, *RR + FT*, shows the slowdown that FastTrack (including RoadRunner) adds to OpenJDK. This slowdown is 9.3X, which is close to the 8.5X slowdown reported by the FastTrack authors [27] in their experiments (with different programs on a different platform).

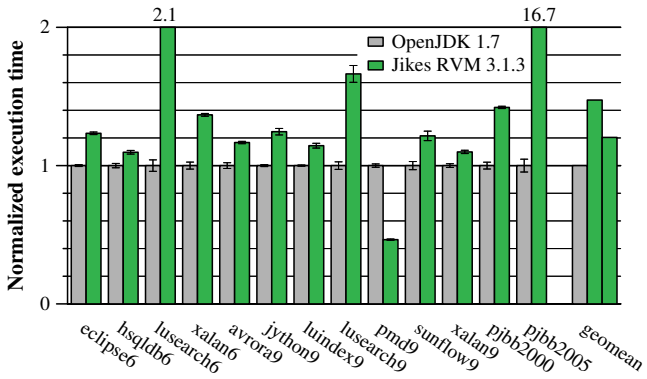
The last two configurations, *Jikes RVM* and *FT (Jikes)*, are the baseline and FastTrack configurations, respectively, from Figure 4. Note that this experiment keeps library instrumentation enabled for the last configuration, *FT (Jikes)*. Our FastTrack implementation in Jikes RVM adds significantly less overhead than the RoadRunner implementation, presumably because the Jikes RVM implementation is *inside* the JVM, so it can add efficient per-field and per-object metadata, modify the garbage collector, and control the compilation of instrumentation. In contrast, RoadRunner is a general framework that is implemented *on top of* the JVM using dynamic bytecode instrumentation.

For these four programs, unmodified Jikes RVM is about 54% slower than unmodified OpenJDK. The next section compares the JVMs' performance across all programs.

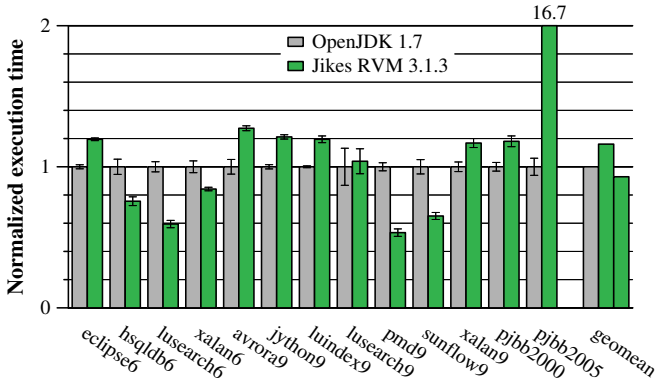
E. Evaluating Competitiveness of Jikes RVM

This section compares the run-time performance of the two JVMs evaluated in Appendix D, Jikes RVM 3.1.3 and OpenJDK 1.7. We use the same configurations as for the *OpenJDK* and *Jikes RVM* configurations from Appendix D. Figure 9 shows the relative performance of OpenJDK and Jikes RVM on two platforms: (a) the 64-core AMD system that most of this paper's experiments use (Section 6 and Appendix D) and (b) the 32-core Intel system used in Appendix C.

As Figure 9 shows, overall Jikes RVM performs competitively with OpenJDK with one significant exception: pjbb2005, which performs 16.7X slower on both platforms, for reasons that are unknown to us. On average, Jikes RVM is 47% and 16% slower than OpenJDK on the AMD and Intel platforms, respectively. Excluding pjbb2005 from the geomean, Jikes RVM is 20% slower (AMD) and 7% faster (Intel) than OpenJDK.



(a) Performance on the 64-core AMD system (Section 6.1).



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

Figure 9. Relative performance of OpenJDK and Jikes RVM on two platforms. In each graph, the two *geomean* bars for Jikes RVM are the geomean including and excluding pjbb2005.

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

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