Efficient, Software-Only Data Race Exceptions

Swarnendu Biswas[†] Minjia Zhang[†] Michael D. Bond[†] Brandon Lucia[§]

[†] Ohio State University
[§] Carnegie Mellon University
{biswass,zhanminj,mikebond}@cse.ohio-state.edu, blucia@cmu.edu

Ohio State CSE Technical Report #OSU-CISRC-3/15-TR04, March 2015

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 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. 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 data races with under 2X slowdown. We also show that Valor is an effective data race detector, providing an appealing cost–coverage tradeoff compared with FastTrack. Overall, Valor advances the state of the art in always-on support for strong behavioral guarantees for data races, and in sound, precise race detection.

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 [40, 53] that may corrupt data, cause a crash, or prevent forward progress [25, 34, 51]. The Northeastern electricity blackout of 2003 [62] is a testament to the danger posed by data races.

The complexity and risk associated with data races provide the two main motivations for this work: (1) systems should provide strong data race semantics that programming languages can rely on, and (2) developers require efficient tools for detecting data races.

The risk presented by data races strongly suggests that programming languages should provide strong semantic guarantees for executions with data races, but doing so efficiently is complex and challenging. As a testament to that complexity, modern languages such as Java and C++ support variants of *DRF0*, providing few or no guarantees for executions with data races [2, 3, 9, 10, 42, 66] (Section 2). These languages remain useful by providing strong semantics with a guarantee of not only SC but also serializability of synchronization-free regions in the absence of data races [3, 11, 42, 43, 53]. Modern languages should provide similarly strong semantics for executions *with* data races.

A promising approach to data race semantics is to detect problematic data races and *halt* the execution with an exception [21, 42, 45]. Exceptional semantics for data races simplify programming language specifications by limiting the effects of executions that exhibit data races. A prerequisite to treating data races as exceptions is having a mechanism that dynamically detects problematic races with no false positives and is efficient enough for always-on use.

Such an efficient, precise data race detector is also valuable for finding and fixing data races, both before and after deployment. During development, high overheads are problematic, wasting limited resources. Developers shy away from using intrusive tools that do not allow them to test realistic program executions [44]. Moreover, the manifestation of a data race is dependent on an execution's inputs, environments, and thread interleavings. A data race may not occur in hours of program execution [69], sometimes requiring weeks to reproduce, diagnose, and fix if it is contingent on specific environmental conditions [29, 40, 62]. Detecting such data races requires analyzing production executions, making performance a key constraint.

Thus, providing strong semantic guarantees and detecting data races each require precise, efficient techniques. *Designing such precise, efficient mechanisms is the central problem we explore in this work.*

Existing approaches. Existing sound 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 [21, 24, 36, 54] (Section 2). Other prior techniques avoid detecting happens-before races soundly (no false negatives), instead detecting conflicts only between operations in concurrent synchronization-free regions (SFRs) [20, 42, 45]. 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, 11, 43]. Unfortunately, existing region conflict detectors are impractical, relying on custom hardware or slowing programs substantially [20, 42, 45].

Our Approach

This work aims to provide a practical, efficient, software-only region conflict detector that is useful for giving data races clear semantics and for reporting real data races. 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 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, but their mechanisms are imprecise, and they do not target providing execution guarantees or detecting data races (Section 8.3).

Alongside Valor, we also developed *FastRCD*, which is an adaptation of the efficient *FastTrack* happens-before detector [24] for region conflict detection. FastRCD does not track the happens-before relation, but it tracks the 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 soundly and precisely tracking the last accesses to each shared variable.

Our purpose in developing FastRCD was to understand the inherent costs of eager conflict detection and FastTrack-like metadata tracking. Our main hypothesis regarding these costs is that tracking the last *read* to each shared variable is inherently expensive. 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.

Both FastRCD and Valor report region conflicts when an access in one thread conflicts with an access that was executed by another thread in an ongoing region—which is a sufficient condition for detecting every data race that could violate region serializability. We show that FastRCD and Valor are still precise even if regions are bounded only at synchronization *release* operations. FastRCD and Valor thus detect conflicts between *release-free regions* (RFRs) by default, which we show helps to amortize per-region costs.

We have implemented FastRCD and Valor, as well as the state-of-the-art FastTrack analysis [24], in a high-performance Java virtual machine. We evaluate and compare the performance, characteristics, and race detection coverage 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 96% average overhead.

By contrast, our implementation of FastTrack adds 342% average overhead over baseline execution, which is comparable to the 8.5X slowdown (750% overhead) reported by prior work [24]—particularly in light of implementation differences (Section 7.2). FastRCD incurs most but not all of FastTrack's costs, adding 283% overhead on average. Valor is not only substantially faster than existing approaches that provide strong semantic guarantees, but its <2X 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 that provides useful, strong semantic guarantees to existing languages for executions both with and without races on commodity systems.

2. Background and Motivation

Detecting data races soundly and precisely enables providing strong execution guarantees, but sound[1] and precise dynamic analysis adds high run-time overhead [24]. Detecting conflicts between synchronization-free regions (SFRs) provides strong execution guarantees, but existing approaches rely on custom hardware [42, 45] or add high overhead [20].

Providing a strong execution model. Modern programming languages (Java and C++) have memory models that are variants of the *data-race-free-0* (DRF0) memory model [3, 11, 43], ensuring data-race-free (DRF) executions are *sequentially consistent* [37] (SC). As a result, DRF0 also provides the stronger guarantee that DRF executions correspond to a serialization of SFRs [2, 3, 42].

Unfortunately, DRF0 provides no semantics for executions with data races; the behavior of a C++ program that permits a data race is undefined [11]. A recent study emphasizes the difficulty of reasoning about data races, showing that a C/C++ program that permits apparently "benign"

[1] A dynamic analysis is sound if it never incurs false negatives *for the current execution*.

data races may behave incorrectly due to compiler transformations or architectural changes [9]. Java attempts to preserve memory and type safety for executions with data races by avoiding "out-of-thin-air" (OOTA) values [43], but recent work shows that it is difficult to prohibit OOTA values without precluding common compiler optimizations [12, 66].

These deficiencies of DRF0 create an urgent need for systems to provide stronger guarantees about data race semantics [2, 10, 16]. Recent work gives *fail-stop semantics* to data races, treating a data race as an *exception* [16, 21, 42, 45]. 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 provides strong execution guarantees by throwing an exception for every data race. However, to detect races soundly and precisely, an analysis must track the *happens-before* relation [36]. Analyses typically track happens-before using vector clocks [47]; 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, to check that each access *happens after* every earlier conflicting access. *FastTrack* reduces the cost of tracking happens-before, yet remains sound (no false negatives), by tracking a single last writer and, in many cases, a single last reader [24].

Despite this optimization, FastTrack still slows executions by nearly an order of magnitude on average [24]. 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, increasing 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 [28].

Hardware extensions (e.g., hardware vector clocks) have the potential to improve performance, but they require invasive architecture changes that are not applicable to today's systems. Furthermore, realistically bounded hardware resources cannot efficiently detect data races that occur over very long spans of dynamic instructions [4, 18, 48].

Our work is motivated by the challenge of designing a precise, software-only data race detector that is efficient enough to be always-on.

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 [20, 42, 45]. SFR conflict detection yields the guarantees that any conflict is a data race; a conflict-free execution is a serialization of SFRs; and a DRF execution produces no conflicts.

Prior work on *Conflict Exceptions* detects conflicts between overlapping SFRs [42] and treats data races as exceptions; its guarantees are the closest to the ones provided by our FastRCD and Valor. 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, the cache coherence mechanism becomes more complex. 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 bars such techniques from 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 [45]. Bounded regions allow DRFx to use simpler hardware than Conflict Exceptions [45, 61], 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 for debugging by detecting conflicts between dynamically overlapping *interference-free regions* (*IFRs*) [20]. An IFR is a region of one thread's execution that is associated with a particular variable, during which another thread's write to that variable is a data race. IFRit comprises both static and dynamic analysis. A whole-program static analysis places 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 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), with extensions to detect and report data races (like IFRit).

3. Efficient Region Conflict Detection

The goal of this work is to develop a region conflict detection mechanism that is useful both for providing guarantees to a programming language implementation and for detecting bugs during development, and is efficient enough for always-on use. We explore two different approaches for detecting region conflicts. The first approach is *FastRCD*, which, like FastTrack [24], uses *epoch optimizations* and *ea-*

gerly detects conflicts at conflicting accesses. Despite being the fastest such mechanism that we are aware of, Section 7.2 experimentally shows that FastRCD's need to track last readers imposes overheads that are similar to FastTrack's and are too high for always-on use.

In response to FastRCD's high overhead, we develop ***Valor***,[2] *which is the main contribution of this work*. Valor detects write–write and write–read conflicts as in FastRCD. The key to Valor is that it detects read–write conflicts *lazily*. Lazy conflict detection is more efficient than eager conflict detection because it need not track last reader information. Instead, 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 those reads and any writes in other threads' ongoing regions. By *lazily* checking for these conflicts, Valor can provide fail-stop data race 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 4 and 5 describe extensions and optimizations for FastRCD and Valor.

3.1 FastRCD: Detecting Conflicts Eagerly in Software

This section presents FastRCD, a novel 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 [42] 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 [24, 36].

FastRCD uses *epoch optimizations* based on FastTrack's optimizations [24] 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 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 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 c@T, where c represents the ongoing region in thread T.

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

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

We generally use T for the current thread and 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 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 conflict with previous writes (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 $\mathcal{W}_x \neq$ epoch(T) then                ▷ Write in same region
2:     let c@t ← $\mathcal{W}_x$
3:     if c = clock(t) then                        ▷ t's region is ongoing
4:         Conflict!                   ▷ Write–write conflict detected
5:     for all t' → c' ∈ $\mathcal{R}_x$ do
6:         if c' = clock(t') then
7:             Conflict!                ▷ Read–write conflict detected
8:     $\mathcal{W}_x$ ← epoch(T)                       ▷ Update write metadata
9:     $\mathcal{R}_x$ ← ∅                               ▷ Clear read metadata
```

At a read, the instrumentation first checks for an earlier read by 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 $\mathcal{R}_x$[T] ≠ clock(T) then             ▷ Read in same region
2:     let c@t ← $\mathcal{W}_x$
3:     if c = clock(t) then                        ▷ t's region is ongoing
4:         Conflict!                    ▷ Write–read conflict detected
5:     $\mathcal{R}_x$[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 at reads and writes each read and write multiple metadata words, the analyses are not amenable to a "lock-free" approach that updates the metadata using a single atomic op-

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

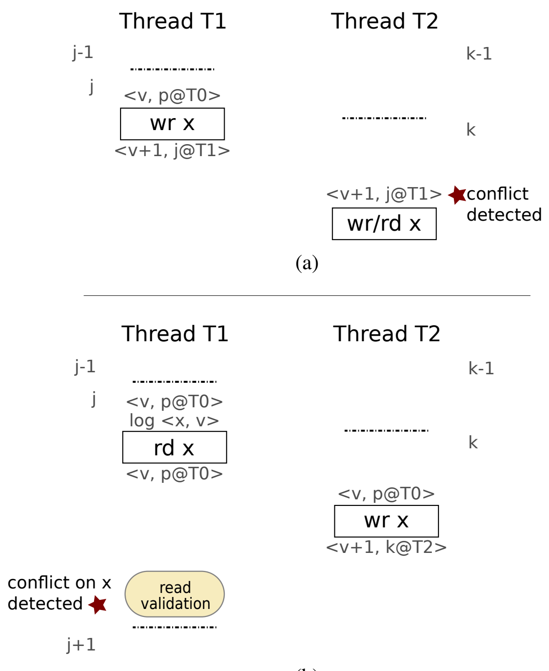


Figure 1. (a) Valor eagerly detects a conflict at T2's access because the last region to write x is ongoing. (b) Valor detects read–write conflicts lazily. During read validation, T1 detects a write to x since T1's read of x.

eration.[3] Note that the analysis and program memory access 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 7.2) because it unavoidably performs expensive analysis at reads.

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, leading to its high performance.

[3] Intel's Haswell architecture supports multi-word read–modify–write via restricted transactional memory (RTM) [68]. Recent work shows that this hardware support incurs substantial per-transaction costs [46, 57].

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 read–write conflicts at the conflicting write. Instead, Valor detects these lazily, when the conflicting read's region ends.

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., $\langle v, p@T0\rangle$) shows x's last writer metadata. Valor stores a tuple $\langle v, c@t\rangle$ that includes the epoch c@t of the last write to x and a *version*, v, that the analysis increments on a region's first write to x. Valor needs versions to detect conflicts precisely, as we explain shortly.

In the example, T1's write to x triggers an update of its last writer metadata to $\langle v+1, j@T1\rangle$. The analysis does not detect a write–write conflict because the example assumes that T0's region p (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 observations imply that T1's ongoing epoch conflicts with T2's access, triggering either a write–write or write–read conflict, depending on the type of T2's access.

Read–write conflicts. In Figure 1(b), T1 reads x, and the last writer metadata remains unchanged because the example assumes that x's last writer was in a completed (i.e., non-conflicting) region in another thread, T0. T1 records its read in its thread-local *read log*. A read log entry, $\langle x, v\rangle$, consists of the address of variable x and x's current version, v.

When T2 writes x, it is a read–write conflict because T1's region j is ongoing. However, the analysis does *not* detect the conflict at T2's write, because Valor does not track x's last readers. Instead, the analysis simply 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 $\langle x, v\rangle$ 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 conflicting write, which Section 3.2.3 argues is acceptable for providing strong behavior guarantees.

Valor requires 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., ⟨x, p@T0⟩ in Figure 1(b). In that case, Valor would still correctly detect the read–write conflict in Figure 1(b). So why does Valor need versions as part of its last writer metadata?

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, even though x's epoch has changed from the value recorded in the read log. Note that the reason there is no conflict, despite the changed epoch, is that 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. 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*,[4] 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. 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 ⟨v, c@t⟩ 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 ⟨x, v⟩, where x is the address of a shared variable and v is a version. Without loss of generality, one can consider it to be a sequential store buffer (allows duplicates) or a set (no duplicates).

As in Section 3.1, we use T for the current thread 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

[4] A region updates x's version only at its first write to x.

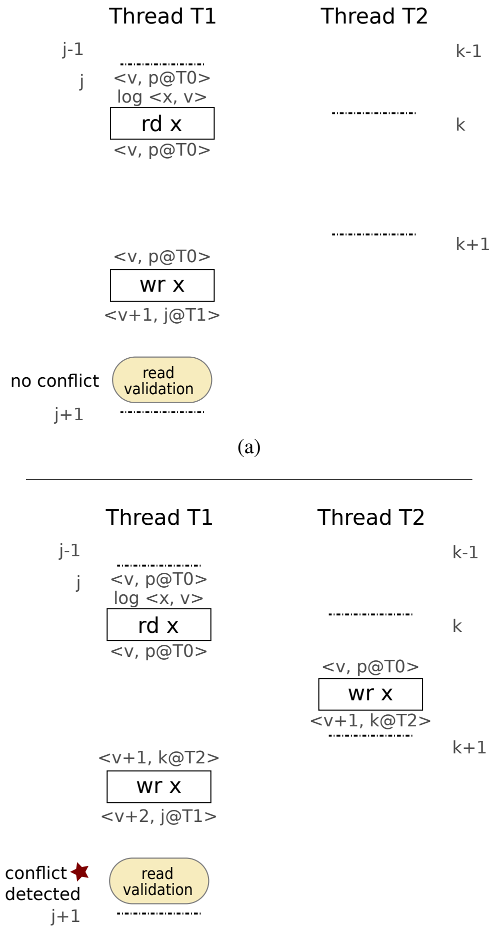


Figure 2. Valor must maintain versions in order to detect conflicts soundly and precisely.

checking if c = clock(t), indicating that x was last written by an ongoing region in another thread (note that this condition 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⟩ ← $\mathcal{W}_x$
2: if c@t ≠ epoch(T) then                        ▷ Write in same region
3:     if c = clock(t) then
4:         Conflict!                 ▷ Write–write conflict detected
5:     $\mathcal{W}_x$ ← ⟨v+1, epoch(T)⟩          ▷ Update write metadata
```

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). After that check, the executing thread adds an entry to its read log (line 4). The new entry consists of x's address and its current version v.

Unlike FastRCD's analysis at reads (Algorithm 2), Valor's analysis at reads does not update any shared metadata, such as FastRCD's read map $\mathcal{R}_x$. FastRCD's updates to read metadata make reads much more expensive, requiring syn-

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

```
1: let $\langle v, c@t \rangle \leftarrow \mathcal{W}_x$
2: if $t \neq T \wedge c = clock(t)$ then
3:     Conflict!                    ▷ Write–read conflict detected
4: $T.readLog \leftarrow T.readLog \cup \{\langle x, v \rangle\}$
```

chronization and incurring remote cache misses on the read metadata, which the original program would not incur.

Analysis at region end. Valor detects read–write conflicts lazily at region boundaries, as shown in Algorithm 5. For each entry $\langle x, v \rangle$ 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 only a difference of *more* than one (i.e., $v' \geq v+2$) indicates a conflict (as illustrated in Figure 2 in Section 3.2.1).

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

```
1: for all $\langle x, v \rangle \in T.readLog$ do
2:     let $\langle v', c@t \rangle \leftarrow \mathcal{W}_x$
3:     if $(v' \neq v \wedge t \neq T) \vee v' \geq v + 2$ then
4:         Conflict!                ▷ Read–write conflict detected
5: $T.readLog \leftarrow \emptyset$
```

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 correctly report such read–write conflicts first, Valor checks for and reports them first, by triggering read validation just before reporting other types of conflicts. As Section 3.2.3 explains, Valor can validate reads at any point to detect outstanding read–write conflicts, and does so before sensitive operations like system calls and I/O.

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 variable, $\mathcal{W}_x$. 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

Valor soundly and precisely detects each access that conflicts with another access executed by an ongoing region. Unlike FastRCD, Valor detects read–write conflicts lazily. Thus, 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 potentially conflicting regions do not become externally visible. To prevent external visibility, Valor validates a region's reads before all sensitive operations, like system calls and I/O. Other conflict and data race detectors have detected conflicts asynchronously [18, 45], providing imprecise exceptions and similar guarantees.

Valor must also handle behavior resulting from a region conflict that would be impossible in any serializable execution. If a region tries to throw an exception that would terminate a region, Valor should first validate the region's reads, throwing a *conflict* exception if it detects a conflict, instead of the program exception. Similarly, Valor must periodically validate a long-running region's reads in case the code is stuck in an infinite loop due to so-far-undetected unserializable behavior. Similar issues apply to software transactional memory systems that use lazy concurrency control and permit so-called "zombie" transactions [32].

4. Detecting Data Races with Valor

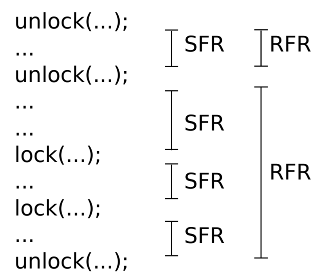
The last section introduced the FastRCD and Valor analyses in terms of arbitrarily defined regions. If the regions are synchronization-free regions (SFRs), then FastRCD and Valor provide a strong execution model that guarantees SFR serializability. In this section, we seek to detect data races using our analyses, by applying them to coarser regions and extending them with source-level debugging information.

4.1 Region Size

Making regions larger helps detect more data races and can potentially help amortize fixed per-region costs. We observe that it is 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). RFR conflicts that are detected by FastRCD and Valor are actual data races.

We note that Valor and FastRCD define a region conflict in the following way: an access executed by one thread that conflicts with an access that was executed 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. Either region conflict definition supports conflict detection between SFRs with no false positives. However, the definition that we use in Valor and FastRCD also supports conflict detection between RFRs without false data races, whereas an overlap-based definition would yield false data races. *We have proved the absence of false data races for our RFR detection scheme* (Appendix A).

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



```
unlock(...);
...
unlock(...);
...
...
lock(...);
...
lock(...);
...
unlock(...);
```

Figure 3. Synchronization- and release-free regions.

statically at compile time. An RFR is at least as large as an SFR, so an RFR conflict detector will detect more conflicts than 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 [24, 54], and some prior work extends redundant instrumentation analysis past acquire, but not release, operations [27].

4.2 Reporting Races

We extend FastRCD and Valor to provide source-level debugging information to improve data race reports. Our goal is to provide the source-level *sites*, including the method and line number, of both accesses involved in a data race.

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 data race, it reports the corresponding recorded site as the first access and reports the current thread's site as the second access. FastRCD similarly records site for every thread–clock entry, and reports the sites for every region conflict.

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

5. Alternate Metadata and Analysis for Valor

Valor maintains version and epoch information to track writes, but doing so has a few disadvantages. Storing versions with epochs increases the space overhead beyond storing epochs alone and increases analysis latency for writes because Valor must update both fields. The metadata should fit in 64 but not 32 bits, which is problematic for 32-bit implementations including ours (Section 6), since metadata updates must be atomic (Section 3.2.2).

This section proposes an alternate metadata representation and algorithm modifications that reduce the space and time costs of storing and accessing Valor's metadata. For clarity, the rest of this section refers to the version of Valor described in Section 3.2 as *Valor-E* (Epoch) and the alternate version introduced here as *Valor-O* (Ownership). We implement and evaluate *Valor-O*.

Versioned writes with ownership tracking. At a high level, Valor must maintain, for each shared variable x, the number of writes to x (i.e., its version), and whether an ongoing region has written x, which Valor-E tracks with a last writer epoch. Valor-O instead tracks whether an ongoing region has written x by maintaining the current "owner thread" of x, which is t if and only if t is currently executing a region that has written x; otherwise it is ϕ. Valor-O maintains a last writer tuple $\langle v, t \rangle$ for each shared variable. The ownership information indicates which ongoing region (if any) has written the variable, and Valor-O uses it to detect conflicts similarly to how Valor-E uses epochs to detect conflicts.

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

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

```
1: let $\langle v, t \rangle \leftarrow \mathcal{W}_x$
2: if $t \neq T$ then                          ▷ Write in same region
3:     if $t \neq \phi$ then
4:         Conflict!                           ▷ Write–write conflict detected
5:     $\mathcal{W}_x \leftarrow \langle v{+}1, T \rangle$      ▷ Update write metadata
6:     $T.\mathsf{writeSet} \leftarrow T.\mathsf{writeSet} \cup \{x\}$
```

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. At a region boundary, the locks for all variables stored in the write set are relinquished.

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 on page 7).

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

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

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

Finally, the analysis 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).

6. Implementation

We have implemented FastTrack, FastRCD, and Valor in Jikes RVM 3.1.3, a high-performance Java virtual machine [6]. Our implementations—which we will make publicly available—share features as much as possible: they instrument the same accesses, and FastRCD and Valor demarcate regions in the same way.

6.1 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.[5]

Instrumenting program operations. The implementations instrument synchronization operations to track happens-before (FastTrack) and 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 4.1) 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), and (3) accesses to a few immutable library types (e.g., String and Integer).

Tracking last accesses and sites. The implementations add last writer and/or reader information to each potentially shared 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. Our evaluation of data race coverage enables site tracking, and our performance evaluation disables tracking of sites.

Eliminating redundant instrumentation. We have implemented an intraprocedural dataflow analysis to identify *redundant* instrumentation points. Instrumentation on an access to a variable is redundant if there is already instrumentation on an access to the same variable earlier in the same region (cf. [14, 27]). Eliminating redundant instrumentation, which we enable by default, reduces the overhead of FastTrack by 3%, FastRCD by 4%, and Valor by 5%.

6.2 FastTrack and FastRCD

Our FastRCD implementation shares many features with our FastTrack implementation, which is faithful to prior work's implementation [24]. 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 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 garbage collection (GC) in two ways. First, every full-heap GC sets a variable's clock to 1 if it was accessed in an ongoing region and to 0 otherwise. Second, every full-heap GC resets each thread's clock to 1. Note that FastTrack cannot use this optimization.

Although we reset clocks at every full-heap GC, a thread's clock may still exceed 22 bits. FastRCD could handle that 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 change the last writer or reader metadata. When metadata must change, the instrumentation ensures atomicity of analysis operations

[5] 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.

by locking one of the shared variable's metadata words using a reserved value set with an atomic operation.

Tracking happens-before. In addition to instrumenting acquire and release synchronization operations as described in Section 6.1, FastTrack tracks the happens-before edge from each static field initialization in a class initializer to corresponding uses of that static field [39]. Our 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.

6.3 Valor

We implement the Valor-O design of Valor described in Section 5.

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 benchmark programs. A version overflow could, with low probability, 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, like how FastRCD resets its clocks (Section 6.2).

Access logging. We implement each per-thread read log as a sequential store buffer (SSB). Valor is tolerant of the duplicate entries inherent in using an SSB. 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, each variable is recorded in a log entry using both its base object address and its 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.

7. Evaluation

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

7.1 Methodology

Benchmarks. We evaluate our implementations of Valor, FastRCD, and FastTrack using large, realistic, benchmarked applications. Our experiments execute the DaCapo benchmarks [7] 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.[6] We omit programs with only one thread or that Jikes RVM cannot execute.

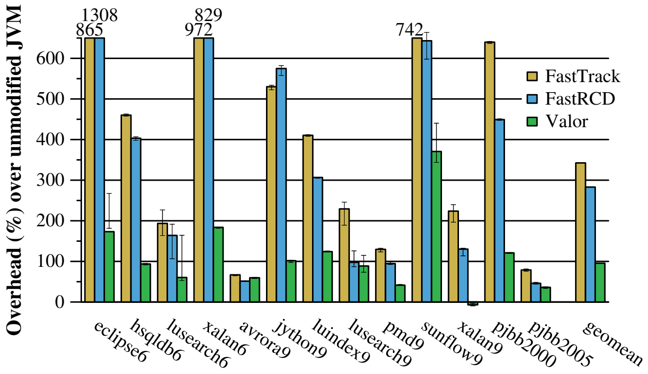


Figure 4. Run-time overhead added to unmodified Jikes RVM by our implementations of FastTrack, FastRCD, and Valor.

Experimental setup. Each detector is built into a high-performance 32-bit JVM configuration that does run-time optimization 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) [19]. 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 B.

7.2 Performance

Figure 4 shows the overhead added over unmodified Jikes RVM by the different implementations. Each bar is the median of 10 trials, in order to minimize the effect of any machine noise. Each bar has a 95% confidence interval that is centered at the mean. *The main performance result in this paper is that Valor incurs only 96% run-time overhead on 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.

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

	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	12,800	3,310	83.5	69.6	64.8	196,000	82
hsqldb6	402	102	621	79	44.8	44.8	30.5	7,600	91
lusearch6	65	65	2,380	798	84.1	84.2	79.4	9,880	322
xalan6	9	9	11,200	2,160	49.8	48.8	23.7	288,000	46
avrora9	27	27	5,470	2,430	90.0	90.1	91.9	6,400	1,230
jython9	3	3	5,220	1,470	63.9	16.0	38.8	199,000	34
luindex9	2	2	316	100	87.3	86.0	71.1	267	1,560
lusearch9	64	64	2,380	693	84.9	85.0	77.3	6,050	508
pmd9	5	5	593	188	86.3	86.1	72.5	2,130	367
sunflow9	128	64	21,700	2,210	95.7	95.7	51.6	9	2,640,000
xalan9	64	64	10,200	2,100	56.1	55.4	28.4	108,000	114
pjbb2000	37	9	1,540	538	37.0	37.4	9.4	128,000	16
pjbb2005	9	9	6,390	2,650	57.3	29.9	9.8	277,000	33

Table 1. Run-time characteristics of our benchmarks and 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%.

FastTrack. Our FastTrack implementation adds 342% overhead on average (i.e., 4.4X slowdown). Prior work reports an 8.5X average slowdown, but for an implementation with many differences with ours [24]. Appendix C describes our empirical comparison of FastTrack implementations to ensure that our FastTrack implementation performs well.

FastRCD. Figure 4 shows that FastRCD adds 283% 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 96% 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 updates to last reader metadata. These updates lock and write metadata word(s), both of which are costly operations.

Valor consistently outperforms FastTrack and FastRCD for all programs except avrora9. As Table 1 shows, avrora9 has an unusually high proportion of reads for which FastTrack and FastRCD do not need metadata updates: they detect these cases efficiently by maintaining last reader metadata. In contrast, Valor logs and validates every read.

7.3 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 our 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 read and write operations that execute (in millions). The *No metadata updates* columns show the percentage of accesses for which instrumentation need not update or synchronize on any metadata. For FastTrack, these are its "same epoch" and "read shared same epoch" cases [24]. For FastRCD and Valor, these are the cases where the analysis does not update any per-variable metadata. For three programs, FastTrack and FastRCD differ significantly in how many reads require metadata updates. The difference exists because the analyses decide differently when to upgrade from a read epoch to a read map (Section 6.2). Minor differences for other programs are not statistically significant.

We report only FastTrack's percentage of per-*write* metadata updates, since FastRCD and Valor usually report very similar percentages. The exception is that FastRCD reports significantly lower percentages for eclipse6. We are investigating this difference.

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.

7.4 Data Race Detection Coverage

We compare the effectiveness of FastTrack, FastRCD, and Valor as data race detectors. FastTrack detects every data race in an execution. FastRCD and Valor detect every data race that is also a region conflict, ensuring region serializability, but they miss races that span large windows of dynamic instructions. Note that FastRCD and Valor are configured to detect conflicts between *release*-free regions (RFRs).

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

	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 in each of 10 trials. The second number (in parentheses) is distinct races reported at least once over all trials.

execution, we count it only once. The first number for each detector is the average number of races reported in each of 10 trials. Run-to-run variation is typically small: 95% confidence intervals are consistently smaller than ±10% of the reported mean, except for xalan9, which varied by ±35% of the mean. The number in parentheses is the count of races reported at least once across all 10 trials.

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 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 [20].

We emphasize that although FastRCD and Valor do not report some data races, the reported races involve accesses that are dynamically "close enough" together to jeopardize region serializability (Section 2). We (and others [20, 42, 45]) 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 write and multiple reads in the same region, whereas FastTrack reports only the read–write race involving the last read by the region.

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 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 96% versus 104% overhead on average, respectively. The difference is likely due to RFRs being larger and thus incurring fewer metadata and write set updates (Section 4.1).

7.5 Summary

Overall, Valor substantially outperforms both FastTrack and FastRCD, adding, on average, just a third of FastRCD's overhead. Valor outperforms FastTrack by 3.6X and FastRCD by 3.0X, respectively. 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 precise dynamic data race detection. FastRCD and Valor detect a substantial fraction of an execution's data races, thus offering a cost–coverage tradeoff that is particularly compelling for Valor. Furthermore, FastRCD and Valor detect every data race that might violate SFR serializability, providing a solid semantic foundation on which to specify a language.

8. Related Work

Section 2 covered the closest related work [20, 21, 24, 42, 45]. This section compares our work to other approaches.

8.1 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) [24]. An alternative is *lockset* analysis, which checks for violations of a locking discipline (e.g., [17, 59, 63]). However, lockset analysis reports false data races, limiting its value for race detection and preventing it from use as a strong execution model. Hybrids of happens-before and lockset analysis tend to report false positives (e.g., [52]).

Goldilocks [21] 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 [24].

Other work gives up soundness, 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 [13, 23, 35, 44]. These approaches incur the costs of tracking the happens-before relation [13, 35, 44] and/or provide limited coverage guarantees [23, 44]. Since they miss many data races, they are limited as race detectors and completely 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 or locksets and extending cache coherence to identify shared accesses [4, 18, 48, 67, 69]. However, manufacturers have been reluctant to change already-complex cache and memory subsystems substantially to support race detection.

Static analysis. Whole-program static analysis has the potential to avoid false negatives, considering all possible program behaviors (e.g., inputs, environments, and thread interleavings) [22, 49, 50, 55, 65]. 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. These limitations make static approaches unsuitable for detecting data races and providing a strong execution model.

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 [17, 21, 38, 64]. 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 6.1). Our 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 or rewriting programs in these languages [8, 56]. Type systems can ensure data race freedom, but they typically require adding annotations and modifying code [1, 15].

8.2 Enforcing Region Serializability

An alternative to detecting violations of region serializability is to *enforce* end-to-end region serializability. Existing approaches either enforce serializability of full synchronization-free regions (SFRs) [53] or bounded regions [5, 60]. They rely on support for expensive speculation that often requires complex hardware support.

8.3 Detecting Conflicts

Software transactional memory (STM) detects conflicts between programmer-specified regions [31, 32]. However, STMs need not detect conflicts precisely; an imprecise conflict merely triggers an unnecessary misspeculation and retry. To avoid the high cost of tracking each variable's last readers, some STMs use so-called "invisible readers" and detect read–write conflicts *lazily* [32]. In particular, *McRT-STM* and *Bartok* detect write–write and write–read conflicts eagerly and read–write conflicts lazily [33, 58]. These mechanisms are thus related to Valor's conflict detection, while fully eager STM conflict detection is analogous to FastRCD's conflict detection. However, these STMs have not introduced designs that provide precise conflicts, nor have they addressed the problems we target: supporting conflict exceptions and useful data race reports.

RaceTM uses *hardware* TM to detect conflicts that are data races [30]. RaceTM is thus closest to existing hardware-based conflict detection mechanisms [42, 45].

Last writer slicing (LWS) tracks data provenance, recording only the last writers of data [41]. 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.

9. Conclusion

This work introduces two new software-based region conflict detectors, one of which, Valor, has overheads low enough to provide semantic guarantees practically to a language specification. The key insight behind Valor is that detecting read–write conflicts lazily retains most semantic guarantees and has better performance than eager data race detection. In addition to its potential for supporting a strong memory model, Valor is an effective data race detector with a compelling cost–coverage tradeoff. Overall, Valor represents an advance in the state of the art for providing strong guarantees and for detecting data races. This advance will help make it practical to use all-the-time data race exceptions and detection in more settings, from in-house testing to alpha and beta testing to even some production systems.

Acknowledgments

We thank Man Cao, Jake Roemer, and Aritra Sengupta for discussions and advice.

A. RFR Conflicts Are Data Races

This section proves the following theorem from Section 4.1:

Theorem. Every RFR conflict is a true data race.

Proof. We prove this claim by contradiction. Let us assume that an RFR conflict exists in a data-race-free 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.

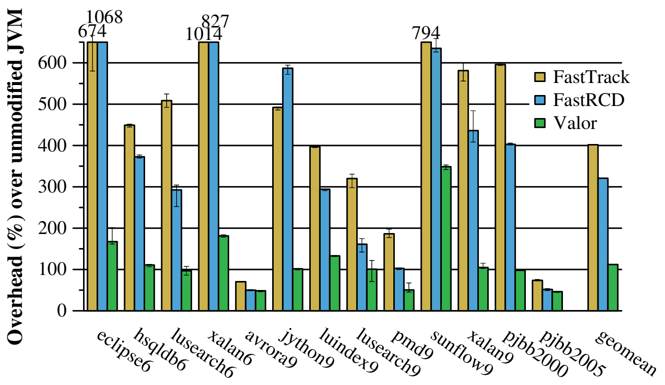
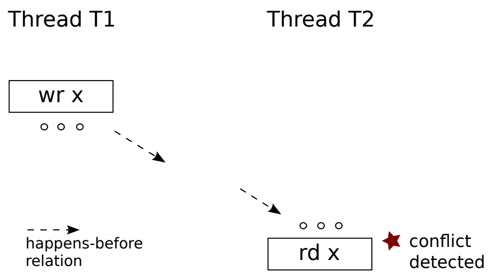


Figure 5. Run-time overhead added to unmodified Jikes RVM by our implementations of FastTrack, FastRCD, and Valor on an Intel Xeon E5-4620 system. Other than the platform, the methodology is the same as for Figure 4.

By the definition of a data race, T1's write *happens-before* T2's read [3, 36], as shown in the following figure:



We represent this happens-before relationship as wr x $\prec_{HB}$ rd x, similar to representations from prior work [43]. The happens-before relation $\prec_{HB}$ is a partial order that is the union of *program* order (i.e., intra-thread order) $\prec_{PO}$ and *synchronization* order $\prec_{SO}$ [36, 43]. Since wr x and rd x execute on different threads, they must be ordered in part by synchronization order, implying that the transitive happens-before relationship consists of the following operations:

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

where rel and acq are synchronization release and acquire operations, e.g., lock operations on the same or different locks.

The rel operation must execute on the same thread as wr x because the operations are ordered only by program order. However, rel ends an RFR, and rel $\prec_{HB}$ rd x, so rd x *happens after* the region that executed wr x has ended. Thus, by the definition of RFR conflict, no RFR conflict exists, which contradicts the initial assumption. □

B. Architectural Sensitivity

We studied 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 7.2. Figure 5 shows the overhead added over unmodified Jikes RVM by our implementations. FastTrack adds an overhead of 402%, while FastRCD adds 321% overhead. Valor continues to substantially outperform the other techniques, adding an overhead of only 112%. The *relative performance* of Valor compared to FastTrack and FastRCD is similar on both platforms. On the Xeon platform, Valor adds 3.6X and 2.9X less overhead than FastTrack and FastRCD, respectively, on average. On the (default) Opteron platform, Valor adds 3.6X and 3.0X less overhead on average.

C. Comparing FastTrack Implementations

To have a fair, direct comparison of FastTrack, FastRCD, and Valor, we have implemented all the three techniques in Jikes RVM (Section 6). It is difficult to reuse the FastTrack implementation from the original work [24] for our performance evaluations since the FastTrack authors' implementation uses the RoadRunner dynamic bytecode instrumentation framework [26], which alone slows programs by roughly 4–5X [24, 26].

To better understand the performance differences between our FastTrack implementation and the original, we compare our implementation of FastTrack in Jikes RVM with the publicly available implementation of FastTrack that is part of the RoadRunner framework[7] [24, 26]. We execute the RoadRunner FastTrack implementation on a different JVM, Open JDK 1.7, because Jikes RVM would not execute it correctly. We have been unable to get RoadRunner to work for most benchmarks, even on OpenJDK, apparently because of RoadRunner limitations related to reflective calls and custom class loading. We have also been unable to enable instrumentation of the Java libraries (e.g., java.*) for similar reasons.

Figure 6 shows how the implementations compare for the subset of programs that work with RoadRunner on OpenJDK. For each program, the first two configuration 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 8.9X, which is close to the 8.5X slowdown reported by the FastTrack authors [24] in their experiments with different programs on a different platform. The last two configurations, *Jikes RVM* and *FT (Jikes)*, are just the baseline and the FastTrack configurations, respectively, from Figure 4. This experiment however disables instrumenting libraries for the last configuration, *FT (Jikes)*, to be consistent with the second configuration, *RR + FT*.

These results show that for these four programs that RoadRunner can execute, Jikes RVM is about 50% slower on average than OpenJDK. However, these four programs are not representative of all programs. We found that across all programs except pjbb2005, Jikes RVM is only 26% slower

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

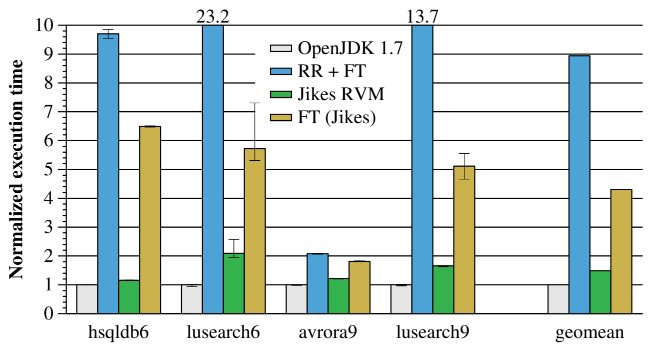


Figure 6. Performance comparison of FastTrack on OpenJDK 1.7 and Jikes RVM. The last two configurations correspond to the baseline and FastTrack configurations in Figure 4.

on average than OpenJDK. However, for pjbb2005, Jikes RVM is 18X slower than OpenJDK, which we are investigating.

Our FastTrack implementation inside a JVM substantially outperforms the prior FastTrack implementation, which is implemented outside of a JVM on top of a general dynamic bytecode instrumentation framework. Overall, this experiment helps to show differences between different platforms and implementations, and it suggests that our FastTrack implementation is competitive with an existing implementation of FastTrack.

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] 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.

[8] 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.

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

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

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

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

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

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

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

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

[28] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, pages 293–303, 2008.

[29] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An Exploratory Survey. In *EC*^{2}, 2008.

[30] 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.

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

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

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

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

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

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

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

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

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

[40] 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.

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

[42] 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.

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

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

[45] 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.

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

[47] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1988.

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

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

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

[51] 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.

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

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

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

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

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

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

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

[65] 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.

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

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

[68] 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.

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