Avoiding Consistency Exceptions Under Strong Memory Models *

Minjia Zhang
Microsoft Research (USA)
minjiaz@microsoft.com

Swarnendu Biswas
University of Texas at Austin (USA)
sbiswas@ices.utexas.edu

Michael D. Bond
Ohio State University (USA)
mikebond@cse.ohio-state.edu

Abstract

Shared-memory languages and systems generally provide weak or undefined semantics for executions with data races. Prior work has proposed memory consistency models that ensure well-defined, easy-to-understand semantics based on *region serializability* (RS), but the resulting system may throw a *consistency exception* in the presence of a data race. Consistency exceptions can occur unexpectedly even in well-tested programs, hurting availability and thus limiting the practicality of RS-based memory models.

To our knowledge, this paper is the first to consider the problem of availability for memory consistency models that throw consistency exceptions. We first extend existing approaches that enforce *RSx*, a memory model based on serializability of synchronization-free regions (SFRs), to avoid region conflicts and thus consistency exceptions. These new approaches demonstrate both the potential for and limitations of avoiding consistency exceptions under RSx. To improve availability further, we introduce (1) a new memory model called *RIx* based on *isolation* of SFRs and (2) a new approach called *Avalon* that provides RIx. We demonstrate two variants of Avalon that offer different performance–availability tradeoffs for RIx.

An evaluation on real Java programs shows that this work's novel approaches are able to reduce consistency exceptions, thereby improving the applicability of strong memory consistency models. Furthermore, the approaches provide compelling points in the performance–availability tradeoff space for memory consistency enforcement. RIx and Avalon thus represent a promising direction for tackling the challenge of availability under strong consistency models that throw consistency exceptions.

CCS Concepts • **Software and its engineering → Runtime environments**

Keywords Memory consistency models; data races

* This material is based upon work supported by the National Science Foundation under Grants CSR-1218695, CAREER-1253703, CCF-1421612, and XPS-1629126.

ISMM'17, June 18, 2017, Barcelona, Spain
ACM. 978-1-4503-5044-0/17/06...
http://dx.doi.org/10.1145/3092255.3092271

1. Introduction

Compilers and architectures for shared-memory parallel programs perform intra-thread optimizations assuming *no* interactions with concurrent threads. To avoid visibly affecting program semantics, these optimizations are restricted from reordering across synchronization operations (e.g., lock acquire and release). Under these restrictions, code regions between synchronization operations appear to execute atomically—but only for program executions that are free of *data races* (conflicting accesses to the same variable that are not ordered by synchronization operations) [2, 4]. However, for executions with data races, compiler and hardware optimizations lead to behaviors that are ill-defined in theory and erroneous in practice [2, 4, 18, 22, 26, 28, 38, 41, 47, 66, 75, 78].

To address this semantic hole, which afflicts programs written in widely used shared-memory languages including C++ and Java [20, 61, 78], researchers have proposed memory models that provide well-defined semantics for *all* program executions [7, 14, 15, 35, 59, 60, 62, 63, 68, 74, 76, 80, 81, 84, 92]. Notable among these models are those that provide *fail-stop* semantics for data races, throwing a *consistency exception* for a data race that may violate consistency guarantees [14, 15, 35, 59, 62, 74, 80, 92] (Section 2). Consistency exceptions seem to be an inherent feature of memory models that provide region serializability, as Section 2 explains.

Problem. In today's systems, data races have undefined behavior, leading to observable errors sporadically (e.g., under a new compiler or a rare thread interleaving). In contrast, in a world where the default is a memory model with fail-stop behavior for data races, programs will have well-defined semantics but might throw consistency exceptions. This situation is analogous to *memory* errors such as buffer overflows, for which unsafe languages such as C++ give undefined behavior, while memory- and type-safe languages such as Java provide fail-stop semantics.

Even well-tested software has data races that manifest under some production environments, inputs, or thread interleavings [52, 69, 85]—which under fail-stop semantics will lead to unexpected consistency exceptions. Unexpected exceptions hurt an execution's *availability* (this paper's term for the duration a program can execute without throwing exceptions), losing some or all of a program's work and thus translating into degradation of performance or quality of service (Section 2).

Our approach. To the best of our knowledge, this paper is the first to consider and address the problem of availability in memory consistency models that generate consistency exceptions. Our first step in this paper is to extend enforcement of an existing strong memory model called *RSx* based on serializability of synchronization-free regions [14, 15, 59], to avoid region conflicts, and thus consistency exceptions, by pausing a thread that is about

to perform a conflicting access. While these extensions improve availability significantly, their efficacy is limited by the need to provide strict serializability

In an effort to improve availability further, we introduce a new memory model called *RIx* based on *isolation* of synchronization-free regions. For an execution with a data race, RIx generates a consistency exception or ensures *isolation* of executing regions—a guarantee based on *snapshot isolation* [11, 36, 39, 50, 55, 56, 70] that is not as strong as strict serializability.

To enforce RIx, we introduce a software approach called *Avalon*. RIx's semantics permit Avalon to *tolerate* read–write conflicts safely, allowing the conflicting write's region to proceed until its end. Interestingly, Avalon provides isolation of regions, or else it *deadlocks* due to a dependence cycle of waiting on conflicts from true data races. Avalon detects such deadlocks and throws a consistency exception.

RIx not only enables an approach (Avalon) that helps avoid consistency exceptions, but it also allows trading some of these avoidance benefits for better performance. We show that Avalon can optionally detect read–write conflicts *imprecisely* (risking false conflicts, but not missing true conflicts) without jeopardizing support for RIx. We leverage this idea to introduce a variant of Avalon, called *Avalon-I*, that represents variables' last reader information imprecisely, enabling lower run-time costs and complexity than the precise variant, which we call *Avalon-P*.

Our evaluation on benchmarked versions of large, real programs shows that Avalon-P, Avalon-I, and our extended RSx-based approaches provide significantly better availability, at competitive cost, than approaches from prior work. Avalon-P provides significantly better availability than the extended RSx-based approaches, at comparable cost. Avalon-I's imprecise conflict detection leads to lower overhead but also lower availability than Avalon-P.

Contributions. This paper makes the following contributions:

- extended designs of RSx-enforcement approaches that provide best-effort avoidance of consistency exceptions;
- a new memory consistency model called *RIx*;
- *Avalon*, a run-time approach that enforces RIx, with two versions that offer different availability–performance tradeoffs;
- a study of RIx's effectiveness at avoiding erroneous behaviors allowed by weak memory models; and
- a performance–availability evaluation that demonstrates new, compelling points in the design space.

2. Background and Motivation

This section motivates the benefits and challenges of memory consistency models that provide fail-stop semantics.

Modern shared-memory languages such as Java and C++ provide variants of the *DRF0* memory model, introduced by Adve and Hill in 1990 [4, 20, 61]. DRF0 (and its variants) provide a strong guarantee for well-synchronized, or *data-race-free*, executions: *serializability of synchronization-free regions* (SFRs) [2, 59].[1] An SFR is a dynamic sequence of executed instructions bounded by synchronization operations (e.g., lock acquires and releases) with no intervening synchronization operations. Every executed non-synchronization instruction is in exactly one SFR. An execution is region serializable if it is equivalent to some serial execution of regions (i.e., some total order of non-interleaved regions).

However, for executions with data races, DRF0 provides weak or no behavior guarantees [2, 18, 19, 21, 22]. C++'s memory model gives undefined semantics for data races [20]. The Java memory model (JMM) provides weak, complex guarantees for executions with data races, in an effort to preserve memory and type safety [61]. However, the JMM precludes common Java virtual machine (JVM) compiler optimizations [78]. In practice, JVMs perform optimizations that violate the JMM [22, 78].

Despite much effort by researchers and practitioners, data races are difficult to avoid, detect, and fix [1, 14, 23, 24, 29, 35, 37, 38, 40, 48, 64, 65, 67, 71, 73, 77, 86, 87, 96]. Data races can manifest only under certain environments, inputs, and thread interleavings [43, 58, 85, 97]. Data races thus occur unexpectedly in production systems, sometimes with severe consequences [52, 69, 85].

Despite the shortcomings of DRF0-based memory models, languages and systems continue to use them in order to maximize performance. DRF0 permits compilers and hardware to perform uninhibited *intra-thread* optimizations within SFRs. Any attempt to provide stronger consistency must consider the impact of restricting optimizations.

Sequential consistency. Much work has focused on providing *sequential consistency* (SC) as the memory consistency model [2, 3, 42, 53, 54, 63, 72, 79, 81, 84].

> An execution is **SC** if its operations appear to interleave in some order that conforms to program order [51].

Enforcing *end-to-end* SC (i.e., SC with respect to the original program) requires restricting optimizations by both the compiler and hardware. (Enforcing SC in the compiler or hardware alone does not provide end-to-end SC.) SC does not necessarily provide a compelling strength–performance tradeoff. Programmers tend to think at a granularity larger than individual memory accesses, Adve and Boehm argue that "programmers do not reason about correctness of parallel code in terms of interleavings of individual memory accesses" [2]. We find that SC does not eliminate some errors due to data races (Section 8.7).

Region serializability. In contrast, memory consistency based on *region serializability* [7, 14, 15, 59, 60, 68, 76] requires fewer or no restrictions on compiler and hardware optimizations. Furthermore, region serializability is strictly stronger than SC. We define *RS* as a property of executions that are *SFR serializable*:

> An execution is **RS** if it is equivalent to a serialization of SFRs.

RS places *no* new restrictions on compiler and hardware optimizations, which already respect synchronization boundaries.

Prior work introduces a memory model, which we call *RSx*, that treats data races as fail-stop errors [59]:

> **RSx** is a memory model that either ensures RS or generates a *consistency exception*—but only if the execution has a data race [14, 15, 59].

Since RSx may or may not throw a consistency exception for an execution with a data race, it permits implementation flexibility: an implementation of RSx does not need to incur the cost and complexity of detecting all data races or RS violations precisely. Prior work provides RSx by detecting *region conflicts* [14, 15, 59], which occur when one region's access conflicts with an access performed by another ongoing region.[2]

Why throw consistency exceptions at all? Why not enforce RS unconditionally without the possibility of exceptions? Some work has enforced RS (without consistency exceptions) [44, 68]. However, enforcing RS efficiently relies on speculative execution,

[1] DRF0 also provides *sequential consistency* (SC) [51] for DRF0 executions. SFR serializability implies SC.

[2] Two memory accesses conflict if they are accesses to the same variable by different threads and at least one is a write.

which adds costs and complexity and cannot easily handle "irrevocable" operations such as I/O [10, 31, 34, 44, 68, 91, 93]. Furthermore, some programs that make progress under SC cannot make progress under RS [76].

Drawbacks of fail-stop semantics. Although RSx provides strong, well-defined semantics, it runs the risk of generating consistency exceptions. In essence, RSx converts DRF0's undefined behavior for data races into fail-stop errors, much like how memory- and type-safe languages handle buffer overflows. If consistency exceptions occur unexpectedly and frequently, they may frustrate developers and users *more than* than today's practical consequences of data races under DRF0.

What happens when a program executing under RSx (or other memory model with fail-stop behavior) generates a consistency exception? The execution could restart from the beginning, or it could resume from a checkpoint. In the case of a server program that handles requests, the execution could abort the current request, relying on the client to resend a request. Consistency exceptions thus degrade performance or quality of service.

Prior work has not considered the issue of availability nor how to reduce exceptions for memory models that generate consistency exceptions, which is *fundamental to the usability and adoptability of these memory models*.

3. Goals and Overview

This paper's goals are to address the challenge of availability and to explore the tradeoff between availability, performance, and semantics in memory consistency models.

Section 4 extends existing *RSx enforcement* approaches to improve availability. Even with these extensions, it seems inherently hard under RSx to avoid consistency exceptions and to achieve high performance. Section 5 introduces a new memory model, called *RIx*, that allows more flexibility to avoid consistency exceptions and to improve performance. Section 6 presents an approach called *Avalon* that enforces RIx, including versions of Avalon that provide different tradeoffs between performance and availability.

4. Increasing Availability Under RSx

This section extends two approaches from prior work that enforce RSx to avoid exceptions by waiting at conflicts. These two approaches, *FastRCD* and *Valor* [14], detect *region conflicts* (conflicting accesses in two concurrent regions) at run time and throw a consistency exception.

FastRCD and FastRCD-A. FastRCD (whose design resembles the design of the FastTrack data race detector [40]), detects region conflicts when they occur [14]. Our extended analysis, called *FastRCD-A* (Available), avoids consistency exceptions by waiting when it detects a region conflict.

At a high level, FastRCD-A detects conflicts by tracking each shared variable's last writer region and last reader region(s). When FastRCD-A detects a conflict, it waits until the conflict no longer exists, thus *avoiding* the conflict.

We present FastRCD-A (and Avalon in Section 6) using the following notation:

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

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

$\mathcal{W}_x$ – Represents last-writer metadata for variable x, as the epoch c@t, which means t last wrote x at time c.

$\mathcal{R}_x$ – Represents last-reader metadata for x, as a *read map* from each thread to a clock value (or 0 if not present in the map).

Algorithm 1 REGION BOUNDARY [FastRCD-A]: thread T reaches region boundary (program synchronization operation)

```
1: incClock(T)
```

Algorithm 2 WRITE [FastRCD-A]: thread T writes x

```
1:  let c@t ← W_x
2:  if c@t ≠ epoch(T) then                          ▷ First write to x by this region?
3:      if t ≠ T ∧ clock(t) = c then                ▷ Write–write conflict?
4:          if deadlocked then
5:              throw consistency exception
6:          else
7:              Retry from line 1
8:      for all t' ↦ c' in R_x do
9:          if t' ≠ T ∧ clock(t') = c' then         ▷ Read–write conflict?
10:             if deadlocked then
11:                 throw consistency exception
12:             else
13:                 Retry from line 1
14:     W_x ← epoch(T)                              ▷ Update write metadata
15:     R_x ← ∅                                     ▷ Clear read metadata
```

Algorithm 3 READ [FastRCD-A]: thread T reads x

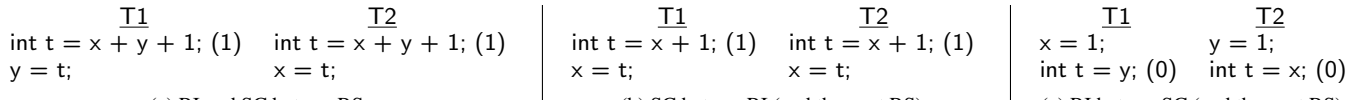
```
1: if clock(T) ≠ R_x[T] then                        ▷ First read to x by this region?
2:     let c@t ← W_x
3:     if t ≠ T ∧ clock(t) = c then                 ▷ Write–read conflict?
4:         if deadlocked then
5:             throw consistency exception
6:         else
7:             Retry from line 1
8:     R_x[T] ← clock(T)                            ▷ Update read metadata
```

Algorithm 1 shows how FastRCD-A increments clock(T) at region boundaries. Algorithms 2 and 3 show FastRCD-A's actions at program reads and writes. The first **if** statement in both algorithms checks whether this region has already written or read this variable, respectively, in which case the algorithm does not need to check for conflicts or update read/write metadata. Otherwise, FastRCD-A checks for write–write and read–write conflicts (lines 3 and 9 in Algorithm 2) and write–read conflicts (line 3 in Algorithm 3) by checking whether the last writer region or reader region(s) are still executing. If FastRCD-A detects a conflict, the current thread T *waits* (by retrying from line 1) for the region (executed by t) to finish its executing region. (In contrast, when FastRCD detects a conflict, it throws a consistency exception immediately [14].)

After checking for (and potentially waiting on) conflicts, the analysis at a write or read updates the variable's write and/or read metadata. An implementation must provide atomicity of instrumentation for each operation, as discussed in Section 7.

FastRCD-A detects *waiting-induced cyclic dependences* by maintaining a global *region wait-for graph* and using an edge-chasing algorithm to find cycles [49]. A detected cycle indicates a deadlock, and FastRCD-A generates a consistency exception.

Valor and Valor-A. As prior work [14] and our evaluation show, FastRCD adds high run-time overhead, which is mainly caused by tracking last-reader metadata in order to detect read–write conflicts exactly. Prior work thus introduces *Valor*, which elides tracking of x's last-reader metadata [14]. Instead, Valor logs each read in a per-thread *read log*, and infers read–write conflicts lazily at region end. We introduce *Valor-A*, which extends Valor to *wait* at detected conflicts instead of throwing a consistency exception. Like FastRCD-A, Valor-A waits at detected write–write and write–read conflicts until the conflict no longer exists or the analysis detects a deadlock



(a) RI and SC but *not* RS. (b) SC but *not* RI (and thus not RS). (c) RI but *not* SC (and thus not RS).

Figure 1. Example executions comparing RI to RS and SC. Each thread executes just one region. Shared variables x and y are initially 0. Values in parentheses are the result of evaluating a statement's right-hand side.

due to waiting on conflicts. However, Valor-A cannot wait when it infers a read–write conflict: the read and write have both already executed, so it is too late to avoid the conflict.

Valor-A (and Valor) thus throw *asynchronous* consistency exceptions for read–write conflicts. Asynchronous exceptions are difficult for programs to handle and for developers to debug. Furthermore, asynchronous exceptions leave regions in an inconsistent state. This problem is particularly acute in the context of an unsafe language such as C++, where so-called "zombie" regions can cause corruption that is not feasible in any RS execution [14, 30, 45].

FastRCD-A and Valor-A only perturb thread interleavings, so they do not introduce behaviors that are not already possible in the original program.

Section 8 shows that although FastRCD-A and Valor-A improve availability (i.e., decrease the rate of consistency exceptions) significantly compared with FastRCD and Valor, their effectiveness is limited by the RSx requirements of serializability and precise conflict detection. This insight leads us to introduce a new memory model that is not as strict as serializability but still provides guarantees at the granularity of synchronization-free regions.

5. RIx: A New Strong Memory Model

This section introduces a new memory consistency model called *RIx*, which is based on providing *isolation* of regions.

5.1 Isolation of SFRs (RI)

We define *isolation of synchronization-free regions* (RI) as a property of executions:

> An execution is **SI** if it satisfies two properties: *write atomicity* and *read isolation* of SFRs.

Database systems have provided these properties in the context of *snapshot isolation* (SI) [11, 36, 39, 70]. RI is essentially equivalent to the typical definition of SI, but the database literature typically defines SI *operationally*, and not all definitions are semantically equivalent. We thus use the new term RI for clarity.

Instead of defining RI precisely, we define *conflict RI*, a sufficient condition for RI that a dynamic analysis can feasibly check on the fly. The rest of this paper uses conflict RI as its working definition of RI. Conflict RI for RI is analogous to *conflict serializability* for serializability [13, 88]. The following definitions are closely based on prior work (on SI for database systems) [5, 6].

A multithreaded execution consists of reads and writes executing in regions (SFRs). The following notation describes read and write operations in an execution:

- $w_i(x)$ — a write to variable x by region R_i
- $r_j(x_i)$ — a read from x by region R_j, which sees the value written by region R_i ($i = j$ is allowed)

The following definition captures the notion of ordering in a multithreading execution:

Definition (Time-precedes order)**.** *The* time-precedes *order $\prec_t$ is a partial order over an execution's operations such that:*

1. *$s_i \prec_t e_i$, i.e., the start of a region i precedes its end.*
2. *For any two regions R_i and R_j, either $e_i \prec_t s_j$ or $s_j \prec_t e_i$. That is, the end of one region is always ordered with start of every other region.*

Note that we use time-precedes order only to *define* conflict RI. To *check* conflict RI at run time, a dynamic analysis does *not* need to compute time-precedes order.

Definition (Conflict RI)**.** *An execution is conflict RI if the following conditions hold:*

1. *Two concurrent regions cannot modify the same variable. That is, for any two writes $w_i(x)$ and $w_j(x)$ such that $i \neq j$, either $e_i \prec_t s_j$ or $e_j \prec_t s_i$.*
2. *Every read sees the latest value written by preceding regions. That is, for every $r_i(x_j)$ such that $i \neq j$:*[3]
   (a) $e_j \prec_t s_i$; and
   (b) for any $w_k(x)$ in the execution such that $j \neq k$, either $s_i \prec_t e_k$ or $e_k \prec_t s_j$.

Note that changing $s_i \prec_t e_k$ (in part 2b) to $e_i \prec_t s_k$ yields a definition for *conflict serializability* [13, 88].

Examples. Figure 1 shows three examples to help understand the differences between RS, RI, and sequential consistency (SC). In each example, suppose that each thread's operations are in a single SFR that executes concurrently with the other thread's SFR. Each example thus has one or more data races. If the regions did not have data races, then only RS behaviors would be possible.

Figure 1(a) shows by example that RI is weaker than RS: the reads see values that are not possible under RS. The database literature calls such behaviors *write skew*; prior work has observed that write skew behaviors are rare in practice [39, 56], and our study of real program errors suggests that such behaviors are rare in shared-memory programs (Section 8.7). (Prior work detects and corrects write skew through dependence graph analysis and "promotion" of reads [27, 56].) Figure 1(b)'s execution violates RI (and thus RS): under RI, the regions cannot appear to execute concurrently because their write sets overlap. While Figure 1(b) shows that SC does not imply (i.e., is not strictly stronger than) RI, Figure 1(c) shows that RI does not imply SC.

Under RI, compilers and hardware can optimize SFRs freely, i.e., they can perform the same optimizations allowed under RSx and DRF0. In contrast, (end-to-end) SC requires restricting compiler and hardware optimizations, e.g., by assuming or inserting memory fences at all memory accesses (Section 2).

Understanding RI. Just as today's programmers mostly ignore or misunderstand memory models, we do not envision programmers needing to reason about RI if memory models by default employ RI semantics. Note that in our RI-based memory model (introduced next), RI semantics can arise only when there is a data race, which should still be considered an error. Nonetheless, expert programmers and system designers may want to understand RI. Intuitively, RI provides write atomicity and read isolation, i.e., writes in an SFR appear as a single operation, and a region's reads see writes from a consistent snapshot, never from concurrently executing regions.

[3] if $i = j$, $r_i(x_j)$ sees the value from the latest $w_i(x_i)$.

A more relevant question than comprehensibility is how well RI, compared with DRF0 and SC and RS, avoids errors caused by data races. Section 8.7 evaluates this question.

5.2 A Memory Model Based on RI

We introduce a new memory model called *RIx* based on RI. Similar to RSx (Section 2), under RIx, an execution with a data race may provide RI or throw a consistency exception if the execution has a data race.

> **RIx** is a memory model that guarantees (1) RS for data-race-free executions and (2) RI or a consistency exception for executions with data races.

Note that RIx relaxes semantics from RS to RI only for executions with data races, which under DRF0 have weak or undefined semantics anyway. For data-race-free executions, RIx provides the same semantics as RSx: serializability of SFRs (and SC).

6. Avalon: Runtime Support for RIx

This section introduces *Avalon*, a novel, software-only approach that provides the RIx memory consistency model.

Overview. Avalon shares some functionality with FastRCD-A (Section 4). Like FastRCD-A, when Avalon detects a *write–write* or *write–read* conflict, it waits until the conflict no longer exists, thus *avoiding* the conflict. However, when Avalon detects a *read–write* conflict, it allows the writing thread to *proceed until the end of its current region*, thus *tolerating* the conflict legally under RIx.

Like FastRCD-A, Avalon detects cycles of waiting dependences and throws a consistency exception. Note that in Avalon, a deadlock must involve at least one write–write or write–read dependence, since Avalon naturally allows regions with mutual read–write dependences to proceed once all of the regions end.

In addition to the notation for FastRCD-A, our presentation of Avalon uses following notation:

T.waitMap – Represents the regions that thread T is waiting on. T.waitMap is a map from a thread t to the latest clock value c of t that executed a read that conflicts with a write in T's current region (or 0 if not in the map).

Region boundaries. To tolerate read–write conflicts, a thread waits at a region boundary, without starting the next region. Avalon thus extends per-thread clocks to differentiate region execution from waiting at a region boundary. In particular, thread T's clock represents two execution states of T:

- clock(T) is *odd* if the region is executing. Note that initially clock(T) is 1.
- clock(T) is *even* if the region has finished executing but is waiting for read–write conflicts at a region boundary.

Algorithm 4 shows how Avalon maintains this invariant by incrementing clock(T) both before and after a region waits for tolerating any remaining read–write conflicts. While clock(T) is even, the algorithm checks whether read–write conflicts still exist; if the reader region is still executing (i.e., if clock(t) = c), the algorithm waits until the reader region finishes executing, throwing a consistency exception if Avalon detects deadlock.

Avalon can detect read–write conflicts either precisely or imprecisely, without risking spurious consistency exceptions. Section 6.1 presents a precise version of Avalon called *Avalon-P* (Precise), and Section 6.2 presents an imprecise version called *Avalon-I* (Imprecise).

Algorithm 4 REGION BOUNDARY [Avalon]: thread T reaches region boundary (program synchronization operation)

```
1: incClock(T)                          ▷ Last region done; not ready to start next region
2: for each t ↦ c in T.waitMap do
3:     while clock(t) = c do            ▷ Read–write conflict still exists?
4:         if deadlocked then
5:             throw consistency exception
6: T.waitMap ← ∅
7: incClock(T)                          ▷ Ready to start next region
```

Algorithm 5 WRITE [Avalon-P]: thread T writes x

```
1: let c@t ← W_x
2: if c@t ≠ epoch(T) then               ▷ First write to x by this region?
3:     if t ≠ T ∧ clock(t) ≤ c + 1 then ▷ Write–write conflict?
4:         if deadlocked then
5:             throw consistency exception
6:         else
7:             Retry from line 1
8:     for all t' ↦ c' in R_x do
9:         if t' ≠ T ∧ clock(t') = c' then   ▷ Read–write conflict?
10:            T.waitMap[t'] ← c'
11:    W_x ← epoch(T)                   ▷ Update write metadata
12:    R_x ← ∅                          ▷ Clear read metadata
```

Algorithm 6 READ [Avalon-P]: thread T reads x

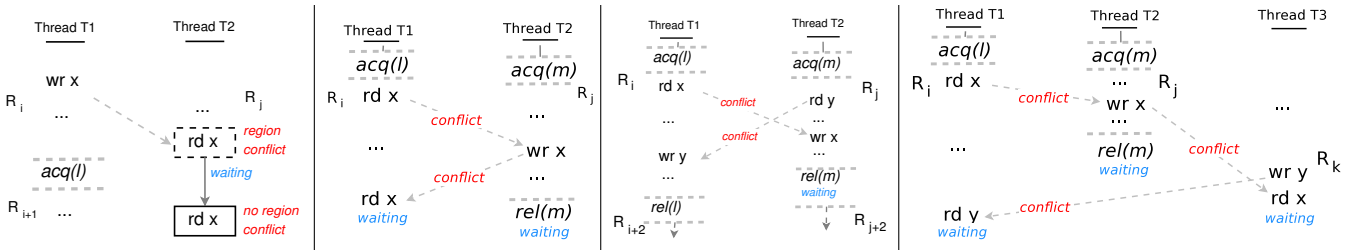
```
1: if clock(T) ≠ R_x[T] then            ▷ First read to x by this region?
2:     let c@t ← W_x
3:     if t ≠ T ∧ clock(t) ≤ c + 1 then ▷ Write–read conflict?
4:         if deadlocked then
5:             throw consistency exception
6:         else
7:             Retry from line 1
8:     R_x[T] ← clock(T)                ▷ Update read metadata
```

6.1 Avalon-P: Detecting Conflicts Precisely

Algorithms 5 and 6 show Avalon-P's analysis at program writes and reads. Like FastRCD-A, if Avalon-P detects a write–write or write–read conflict, the current thread T waits for the writer region (executed by t) to finish any waiting at its region boundary. In order to take into account the two increments to clock(t) at a region boundary, T checks if the writer thread t's clock is at least *two* greater than the variable's clock c, i.e., clock(t) $\geq$ c + 2. In contrast with FastRCD-A, when Avalon-P detects a read–write conflict (line 9 in Algorithm 5), instead of waiting, the current thread T records the conflicting thread t' and its current clock c' in T.waitMap.

Examples. To help understand Avalon-P's algorithm, Figure 2 shows examples of how Avalon enforces RIx. The figure applies to both Avalon-P and Avalon-I, which behave the same for each of the examples. In the figure, concurrent regions access the shared variables x and y. The gray dashed lines (around synchronization operations, e.g., acq(l) and rel(l)) indicate SFR boundaries. R_i and R_j are SFR identifiers, where i and j are per-thread clocks for the respective threads.

In Figure 2(a), T2 tries to read x, which is a region conflict with the previous write to x by T1 in region R_i, because T1 is still executing R_i. T2 handles the region conflict by waiting until T1 finishes its region (R_i), at which point T2 retries its read at time (2) and continues execution safely. In Figure 2(b), T2 waits on the read–write conflict at its region boundary, but T1 is unable to make progress due to a cyclic dependence. In Figure 2(c), a cyclic



(a) Avalon avoids region conflicts. (b) Avalon deadlocks. (c) Avalon tolerates region conflicts. (d) Avalon deadlocks.

Figure 2. Examples showing how Avalon works. (The examples applies to both Avalon-P and Avalon-I.) Dashed lines indicate where Avalon increments its clock. The exact synchronization operations (e.g., acq(m) versus rel(l)) are arbitrary and not pertinent to the examples.

dependence exists, but each region can reach its region boundary, allowing each to proceed. In Figure 2(d), Avalon deadlocks due to a cycle of transitive dependences involving two variables. Each thread gets stuck waiting: T1 and T3 at accesses, and T2 at a region boundary.

6.2 Avalon-I: Leveraging Approximation for Performance

Tolerating read–write conflicts allows Avalon-P to avoid exceptions encountered by prior work, but tracking each variable's last readers precisely is expensive (as prior work also finds [14]). Based on this observation, this section describes *Avalon-I*, a design that *over-approximates last readers*.

The key insight of having imprecise readers is that a thread can wait to perform a write that is *potentially* involved in a read–write conflict. Importantly, Avalon-I still provides RIx: although Avalon-I may wait on a false read–write conflict, any deadlock must include at least one (true) write–read or write–write conflict (this property also holds for Avalon-P), indicating a data race.

In the design of *Avalon-P* from Section 6.1, each variable x has metadata both for writes ($\mathcal{W}_x$) and reads ($\mathcal{R}_x$). Both are needed for precise tracking of last reader(s). In particular, the read metadata is inflated into a read map when there are multiple concurrent reader regions. Since Avalon-I does not require precise detection of read–write conflicts, it allows for a more efficient design.

Metadata compression. Like Avalon-P, Avalon-I maintains the epoch of the last writer for each shared variable x. *Unlike* Avalon-P, Avalon-I maintains precise last-reader information *only* if there exists a single reader; for multiple readers, it abandons precise tracking (so any ongoing region is a potential reader). As a result, Avalon-I represents a variable's last writer and reader information as a single unit of metadata $\mathcal{M}_x$ that always has one of the following values:

- $\mathsf{WrEx}_{\mathtt{c@t}}$ — x was last accessed by region c@t, and that region performed a write to x.
- $\mathsf{RdEx}_{\mathtt{c@t}}$ — x was last accessed by region c@t, and that region performed only reads to x.
- RdSh — At some point since the last write to x, there were multiple concurrent reader regions of x. Any ongoing region may have read x, but no ongoing region may have written x.

The first write to x in region c@t updates $\mathcal{M}_x$ to $\mathsf{WrEx}_{\mathtt{c@t}}$. Similarly, the first read to x in c (if there is no prior write in the same region) updates $\mathcal{M}_x$ to $\mathsf{RdEx}_{\mathtt{c@t}}$. If a second read from a different thread reads the variable while the first read's region is still ongoing, Avalon-I changes $\mathcal{M}_x$ from $\mathsf{RdEx}_{\mathtt{c@t}}$ to RdSh. If $\mathcal{M}_x$ can be encoded in a single word, a single atomic instruction is sufficient to update it (Section 7).

Avalon-I's analysis. Algorithms 7 and 8 show the analysis that Avalon-I performs at each program write and read. In Algorithm 7,

Algorithm 7 WRITE [Avalon-I]: thread T writes x

```
1:  if M_x ≠ WrEx_{c@T} | c = clock(T) then
2:                                          ▷ First write to x by region?
3:      if M_x = WrEx_{*@t} | t ≠ T then
4:          if clock(t) ≤ c + 1 | M_x = WrEx_{c@t} then
5:                                          ▷ Write–write conflict?
6:              if deadlocked then
7:                  throw consistency exception
8:              else
9:                  Retry from line 1
10:     else if M_x = RdEx_{*@t} | t ≠ T then
11:         if clock(t) = c | M_x = RdEx_{c@t} then
12:                                         ▷ Potential read–write conflict?
13:             T.waitMap[t] ← clock(t)
14:     else if M_x = RdSh then
15:         for each thread t ≠ T do        ▷ Potential read–write conflicts
16:             T.waitMap[t] ← clock(t)
17:     M_x ← WrEx_{c@T} | c = clock(T)
```

Algorithm 8 READ [Avalon-I]: thread T reads x

```
1:  if M_x ∉ {RdEx_{c@T}, WrEx_{c@T}, RdSh} | c = clock(T) then
2:                                          ▷ First access to x by region?
3:      M'_x ← RdEx_{c@T} | c = clock(T)
4:      if M_x = WrEx_{*@t} | t ≠ T then
5:          if clock(t) ≤ c + 1 | M_x = WrEx_{c@t} then
6:                                          ▷ Write–read conflict?
7:              if deadlocked then
8:                  throw consistency exception
9:              else
10:                 Retry from line 1
11:     else if M_x = RdEx_{*@t} | t ≠ T then
12:         if clock(t) = c | M_x = RdEx_{c@t} then
13:                                         ▷ Concurrent reader?
14:             M'_x ← RdSh
15:     M_x ← M'_x
```

if the last write to x comes from the same region, the current write can skip the rest of the analysis operations (line 1) since $\mathcal{M}_x$ not need to be updated. If the last write is from the same thread T, the write can update $\mathcal{M}_x$ with the epoch of the current region (R_c). Otherwise, T handles $\mathsf{WrEx}_{\mathtt{*@t}}$ and $\mathsf{RdEx}_{\mathtt{*@t}}$ (* denotes "any clock value") similarly to Avalon-P, by detecting a write–write or a read–write conflict (lines 3–13). If $\mathcal{M}_x$ is RdSh, Avalon-I conservatively assumes read–write conflicts with other threads' ongoing regions (lines 14–16).

In Algorithm 8, if the same region has already read or written x or $\mathcal{M}_x$ is RdSh, T does not need to update $\mathcal{M}_x$ (line 1). If the read is the first read to x in a region before any writes to x, the read updates $\mathcal{M}_x$ from $\mathsf{WrEx}_{\mathtt{c'@t}}$ to $\mathsf{RdEx}_{\mathtt{c@T}}$, so that a write from

	Threads		Memory accesses		Region conflicts			Dyn.	Avg. accesses
	Total	Max live	Reads	Writes	Write–write	Write–read	Read–write	SFRs	per SFR
eclipse6	18	12	4,500M	1,400M	0	3.2K	21	150M	40
hsqldb6	402	102	250M	31M	0	33	1.9	11M	25
lusearch6	65	65	1,100M	400M	0	96	0	9.9M	150
xalan6	9	9	990M	220M	1.4K	520	26	58M	21
avrora9	27	27	900M	440M	380K	3.4M	25K	3.9M	350
jython9	3	3	720M	230M	0	0	0	100M	9.2
luindex9	2	2	290M	97M	0	0	0	540K	720
lusearch9*	32	32	1,100M	350M	38	5.7K	22	7.2M	210
pmd9	5	5	290M	97M	6.6K	5.3K	120	2.4M	160
sunflow9*	64	32	6,700M	720M	0	3.9	4.9	16K	450K
xalan9*	32	32	940M	210M	200	1.4K	42	22M	53

Table 1. Runtime characteristics of the evaluated programs, rounded to two significant figures. *Three programs support varying the number of active application threads; by default, this value is equal to the number of cores (32 in our experiments).

a different thread can still detect a read–write conflict precisely at a $\mathsf{WrEx}_{\mathtt{c'@t}} \rightarrow \mathsf{WrEx}_{\mathtt{c@T}}$ transition instead of having a potential read–write conflict (an alternative is to change $\mathcal{M}_x$ to RdSh, but it would lead to unnecessary imprecision and more consistency exceptions).

7. Implementation

We have implemented FastRCD-A, Valor-A, Avalon-P, and Avalon-I in Jikes RVM 3.1.3 [8, 9], a JVM that performs competitively with commercial JVMs [14]. Our FastRCD-A and Valor-A implementations extend publicly available Jikes RVM implementations of FastRCD and Valor [14]. We have made our implementations publicly available on the Jikes RVM Research Archive.[4]

Our implementations, which target IA-32,[5] extend Jikes RVM's just-in-time compilers to insert instrumentation at synchronization operations and memory operations. The implementations transform the same set of memory operations (field and array element accesses in the application and libraries), demarcate regions in the same way (at lock, monitor, thread, volatile operations, and other synchronization such as atomic accesses), and reuse code as much as possible. Java supports reentrant locks; the implementations demarcate regions at critical sections on reentered locks, but an implementation could instead safely ignore these operations.

We note that all of the implementations apply to existing, unmodified Java programs. The implementations demarcate regions at existing program synchronization operations. They do *not* require any annotations from programmers. They do *not* remove or disable any synchronization operations; even under RIx or RSx, programs still need their lock operations to ensure mutual exclusion of critical sections.

Avalon-P and FastRCD-A add two words of per-variable metadata for tracking writes and reads. Each variable's read metadata can be inflated to a pointer to a read map, and the map's space overhead is proportional to the number of reader threads. In contrast, Avalon-I uses a single metadata word per variable: 21 bits for the clock, 9 bits for the thread ID, and 2 bits for encoding the state (WrEx vs. RdEx vs. RdSh).

Avalon-I safely resets clocks to 0 or 1 at full-heap garbage collection to avoid overflow (adapting prior work's optimization [14]).

Instrumentation atomicity. Avalon-P and FastRCD-A use two words or more of per-variable metadata (to support a read map). The instrumentation initially "locks" one of the variable's metadata words (using an atomic operation and spin loop) and later "unlocks" it (using a store and memory fence) when updating the metadata. The instrumentation performs no synchronization when it performs no metadata updates (for a read or write in the same region).

In contrast, Avalon-I and Valor-A use a single word of shared metadata per variable, so they provide lock-free instrumentation atomicity, using a single atomic operation for updates.

8. Evaluation

This section measures availability, performance, scalability, space usage, and other characteristics for Avalon and competing approaches that provide RSx and RIx.

8.1 Methodology

Benchmarks. Our evaluation uses benchmarked versions of large, real applications: the DaCapo benchmarks with the default workload size, versions 2006-10-MR2 and 9.12-bach (distinguished with names suffixed by 6 and 9) [16]. We omit single-threaded programs and programs that Jikes RVM 3.1.3 cannot execute.

Experimental setup. For each implementation (FastRCD-A, Avalon-P, etc.), we build a high-performance configuration of Jikes RVM that adaptively optimizes the code and uses the default, high-performance, generational garbage collector [17], which adjusts the heap size automatically at run time. Each performance result is the mean of 25 trials. Each reported statistic is the mean from 10 trials of a special statistics-gathering configuration. For each result, we also report a 95% confidence interval.

Platform. The experiments execute on an Intel Xeon E5-4620 machine with four 8-core processors (32 cores total), running Red-Hat Enterprise Linux 6.7, kernel 2.6.32.

Run-time characteristics. Table 1 shows characteristics of the evaluated programs that are independent of the implementation (e.g., Avalon-P vs. FastRCD-A). The *Threads* columns report both threads created and maximum threads active at any time. *Memory accesses* are executed program loads and stores of fields and array elements, which all implementations instrument.

The *Region conflicts* columns show how many of each kind of region conflict occur. Conflicts vary significantly in count and type across programs, but they are generally many orders of magnitude smaller than total memory operations, except for avrora9, which incurs millions of dynamic conflicts. We find that these conflicts are due to 14 *static* data races (distinct unordered pairs of static program locations).

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

[5] Jikes RVM provides robust backend support for IA-32 but not yet x86-64.

	FastRCD [14]	**Valor** [14]	**FastRCD-A**	**Valor-A**	**Avalon-P**	**Avalon-I**
eclipse6	**6.8K** ± 10K	**260** ± 510	**0.2** ± 0.5	**0.2** ± 0.3	**0.3** ± 0.6	**1.2** ± 1.6
	(**14K** ± 24K)	(**1.8K** ± 3.5K)	(**0.2** ± 0.5)	(**0.7** ± 0.9)	(**0.3** ± 0.6)	(**1.2** ± 1.6)
hsqldb6	**27** ± 2.7	**73** ± 2.6	**0.8** ± 0.5	**0.6** ± 0.3	**0.1** ± 0.1	**1.0** ± 0.5
	(**53** ± 5.6)	(**140** ± 5.4)	(**0.8** ± 0.5)	(**0.6** ± 0.3)	(**0.1** ± 0.1)	(**1.2** ± 0.6)
lusearch6	**0.1** ± 0.1	**0.2** ± 0.2	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0
	(**0.1** ± 0.1)	(**0.2** ± 0.2)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)
xalan6	**48** ± 1.7	**53** ± 1.7	**12** ± 1.3	**25** ± 1.1	**5.5** ± 1.4	**10** ± 1.3
	(**120** ± 5.8)	(**93** ± 3.4)	(**12** ± 1.3)	(**39** ± 1.1)	(**5.5** ± 1.5)	(**11** ± 1.4)
avrora9	**200K** ± 3.1K	**230K** ± 2.6K	**38K** ± 760	**28K** ± 400	**16K** ± 430	**18K** ± 430
	(**610K** ± 6.4K)	(**670K** ± 9.7K)	(**39K** ± 820)	(**36K** ± 920)	(**24K** ± 730)	(**27K** ± 760)
jython9	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0
	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)
luindex9	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0	**0** ± 0
	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)
lusearch9	**63** ± 9.7	**62** ± 7.3	**62** ± 7.6	**13** ± 3.3	**20** ± 3.7	**19** ± 5.7
	(**110** ± 25)	(**93** ± 13)	(**62** ± 7.6)	(**13** ± 3.3)	(**21** ± 4.5)	(**21** ± 6.6)
pmd9	**450** ± 150	**370** ± 150	**91** ± 10	**71** ± 7.2	**52** ± 9.5	**77** ± 7.2
	(**3.3K** ± 520)	(**3.5K** ± 930)	(**93** ± 11)	(**170** ± 62)	(**110** ± 51)	(**120** ± 17)
sunflow9	**6.1** ± 1.8	**11** ± 3.6	**0** ± 0	**0** ± 0	**0** ± 0	**4.6** ± 1.1
	(**23** ± 6.3)	(**37** ± 14)	(**0** ± 0)	(**0** ± 0)	(**0** ± 0)	(**4.7** ± 1.2)
xalan9	**330** ± 35	**3.0K** ± 490	**4.7** ± 3.7	**40** ± 3.6	**0.3** ± 0.4	**16** ± 3.2
	(**1.5K** ± 220)	(**6.2K** ± 1.2K)	(**4.7** ± 3.7)	(**40** ± 3.6)	(**0.3** ± 0.4)	(**19** ± 4.5)

Table 2. The number of consistency exceptions reported by implementations that provide RSx and RIx, with 95% confidence intervals. For each program, the first row is dynamic regions that report at least one exception, and the second row (in parentheses) is dynamic exceptions reported. Reported values are rounded to two significant figures (except for values < 1).

The last two columns of Table 1 report regions executed and average region size (i.e., memory operations executed in each SFR). All programs except sunflow9 perform synchronization at least every 720 memory operations on average.

8.2 Availability

The programs we evaluate have *not* been developed or debugged under the assumption that data races are (fail-stop) errors. They have data races that frequently manifest as region conflicts. Under an RSx- or RIx-by-default assumption, programmers would fix frequently occurring data races, but other data races would remain. By comparing how many consistency exceptions each implementation throws, we can estimate its relative ability to avoid exceptions if programs had been developed and debugged to mostly avoid consistency exceptions.

Table 2 reports consistency exceptions thrown by all implementations. The implementations do not actually generate real exceptions; rather, they simply report the exception and allow execution to proceed. In practice, a system could handle consistency exceptions by terminating the execution, by restarting the execution, or resuming from a checkpoint (Section 2). Thus, generating a consistency exception is undesirable but not unacceptable, and decreasing the rate of consistency exceptions is desirable.

FastRCD and Valor report an exception whenever they detect a conflict. FastRCD-A, Valor-A, Avalon-P, and Avalon-I report an exception whenever they detect a deadlock. Valor and Valor-A also report an exception whenever they infer a conflict from a read validation failure.

An executed region might execute multiple, possibly related, conflicting accesses. To downplay this effect, the first row for each program is the number of dynamic regions that report at least one consistency exception. The second row is the number of consistency exceptions reported during the program execution.

A key benefit of this paper's approaches is their ability to improve the availability of memory models with fail-stop semantics. Comparing with prior work's FastRCD and Valor, FastRCD-A, Valor-A, Avalon-P, and Avalon-I report substantially fewer consistency exceptions. Compared with FastRCD and Valor, Avalon-P reduces exceptions by up to three orders of magnitude (e.g., eclipse6, hsqldb6, xalan6, pmd9, and xalan9), or eliminates consistency exceptions completely (e.g., lusearch6 and sunflow9).

Avalon's ability to reduce consistency exceptions comes from two sources: (1) avoiding write–write and write–read conflicts and (2) tolerating read–write conflicts (by exploiting RIx). To analyze the effect of the first feature alone, we can compare FastRCD-A with FastRCD, and Valor-A with Valor. Waiting at conflicts is generally effective at reducing consistency exceptions, reducing exceptions significantly for most programs, albeit not as much as for Avalon-P. This evaluation of waiting at conflicts—which to our knowledge is the first such evaluation—suggests that this technique is generally effective at avoiding consistency exceptions while still preserving the consistency model (whether RSx or RIx).

To analyze the effect of the second feature—tolerating read–write conflicts by exploiting RIx's properties—we can compare Avalon-P with FastRCD-A. As expected, the table shows that Avalon-P generally avoids more consistency exceptions than FastRCD-A. Avalon-P has fewer exceptional regions than FastRCD-A for six programs, and never has more exceptional regions than FastRCD-A (for eclipse6, the difference is not statistically significant).

Relaxed precision. A separate aspect of Avalon is that it permits relaxing precision of conflict detection, potentially improving performance, but at the cost of potentially more consistency exceptions. To evaluate this *cost*, we compare reported consistency exceptions for Avalon-P and Avalon-I. Unsurprisingly, Avalon-I reports more exceptions than Avalon-P, since Avalon-I introduces waiting at region boundaries for spurious read–write conflicts. This effect is mixed across programs: for six programs, Avalon-I generates more exceptions than Avalon-P (although the increases are often modest), whereas we find no statistically significant difference for the other five programs.

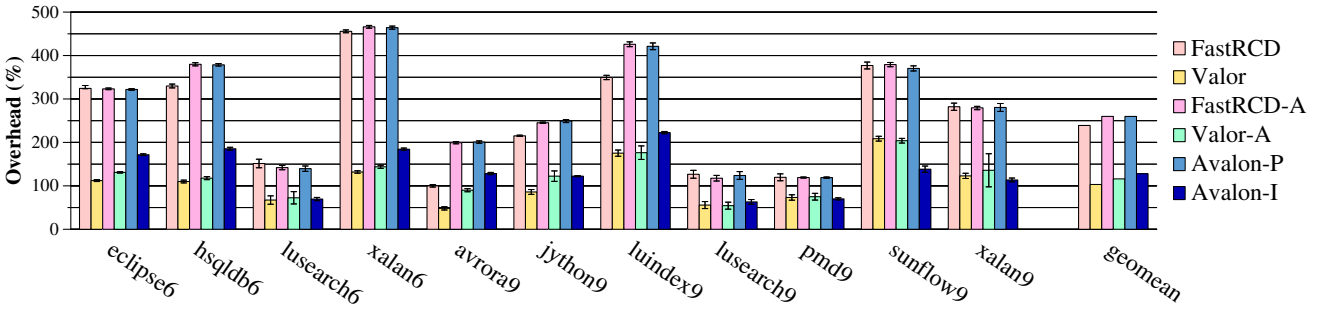


Figure 3. Run-time overhead of prior work's FastRCD and Valor [14] and our FastRCD-A, Valor-A, Avalon-P, and Avalon-I.

In summary, Avalon increases availability significantly over existing approaches that support RSx for all benchmarks. The improvement is particularly significant for applications for which existing approaches generate hundreds or thousands consistency exceptions, while Avalon generates a few or no exceptions. Waiting at conflicts and providing RIx (instead of RSx) both contribute to Avalon's availability improvement over prior work. Relaxing Avalon's precision decreases availability, but perhaps not excessively. Of course, each approach's suitability is a function of not only availability, but also performance, which we evaluate next.

8.3 Performance

This section measures and compares the performance of the RSx- and RIx-enforcing implementations. Since executions are multithreaded, performance overheads include not just instrumentation overhead but also time spent on waiting, which differs among approaches that use different waiting conditions. Section 8.5 tries to isolate this cost by varying application thread counts.

Figure 3 shows run-time overhead added over execution on an unmodified JVM. Our results are in agreement with prior work, which shows that FastRCD adds high run-time overhead in order to maintain last readers, while Valor incurs significantly lower overhead by logging reads locally and validating them lazily [14]. FastRCD-A and Valor-A each add modest overhead over FastRCD-A and Valor-A, respectively, by waiting at conflicts.

Avalon-P tracks write and read metadata in the same way as FastRCD and FastRCD-A, so unsurprisingly it incurs similar overhead. However, Avalon-P incurs slightly *less* overhead than FastRCD-A because Avalon-P avoids waiting at read–write conflicts (instead defers waiting until region end; Section 6). Still, Avalon-P adds 260% overhead on average in order to provide precise conflict detection. In contrast, Avalon-I enables a significantly faster analysis that adds 128% overhead on average, which is competitive with the fastest approach, Valor.

To analyze the performance difference between Avalon-I and Avalon-P, we implemented a configuration (not shown in the figure) that tracks both read and write metadata similarly to Avalon-P (i.e., separate write and read metadata words), but does not track multiple readers precisely, representing multiple readers using a simple "read shared" state. On average this configuration incurs 75% of the overhead that Avalon-P incurs over Avalon-I, suggesting that most but not all of Avalon-I's performance advantage comes from being able to compress its metadata in a single metadata word and use a lock-free update, leading to significantly cheaper instrumentation.

To isolate Avalon-I's overhead due to waiting at conflicts (versus instrumentation overhead), we evaluated an Avalon-I configuration, which we call *Avalon-I-no-wait* (not shown in the figure), that does not wait at conflicts, but instead allows a thread to proceed immediately after detecting a region conflict. Compared with *Avalon-I-no-wait)*, default Avalon-I adds only 9.5% additional overhead (relative to baseline, unmodified execution), suggesting that little of Avalon-I's overhead is due to waiting at conflicts. This result makes sense because conflicts are orders of magnitude smaller than total memory operations (Table 1). That is, run-time overhead is dominated by detecting memory access conflicts.

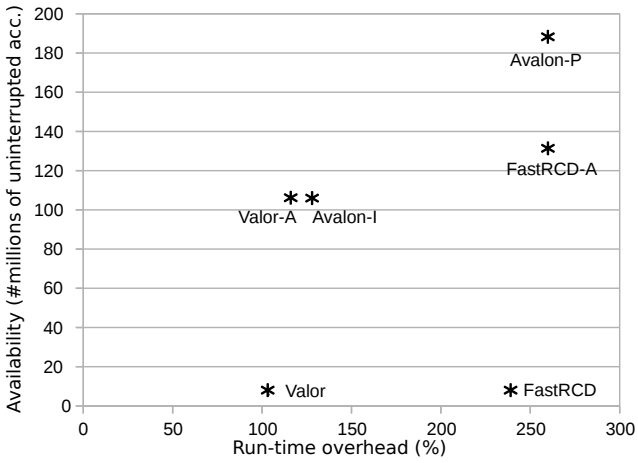


Figure 4. Availability vs. performance for all implementations.

All of our exception-avoiding implementations perform close to the state-of-the-art software-only approaches that support RSx. On average, FastRCD-A and Avalon-P's run-time overheads are within 9% of FastRCD's overhead. The run-time overheads of Valor-A and Avalon-I are within 24% of Valor's overhead on average.

8.4 Performance–Availability Tradeoff

In order to evaluate availability and performance together, Figure 4 plots the previous availability and performance results. The x-axis is the geomean of run-time overhead across all programs. The y-axis is availability, which is defined as the geomean of memory accesses performed without interruption by a consistency exception. That is, for each program execution,

$$availability = \frac{\#\ memory\ accesses\ executed}{\#\ consistency\ exceptions + 1}$$

Larger values represent better availability. Points closer to the top-left corner represent a better performance–availability tradeoff.

Valor performs best, but provides worse availability than FastRCD-A, Valor-A, Avalon-P, and Avalon-I. Avalon-P has the best availability, but its performance overhead is relatively high. Avalon-I and Valor-A arguably each offer the best tradeoff between availability and performance. Although Valor-A provides a stronger

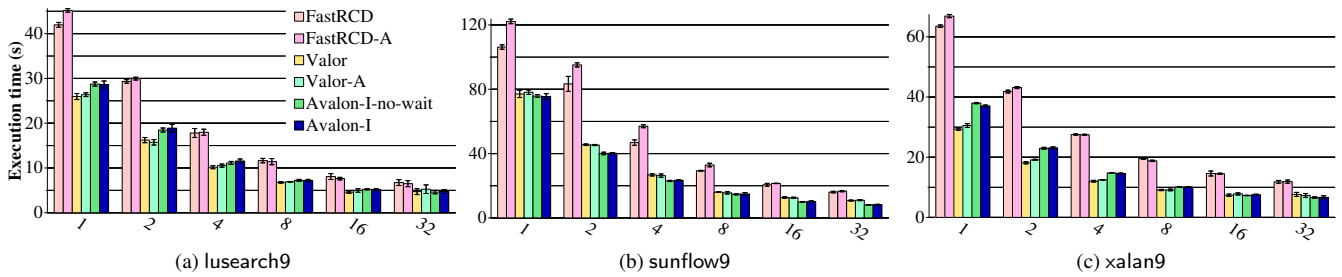


Figure 5. Comparison of execution time for implementations that can wait at conflicts (FastRCD-A, Valor-A, Avalon-I) and implementations that do not wait (FastRCD, Valor, Avalon-I-no-wait), for 1–32 application threads. The legend applies to all graphs.

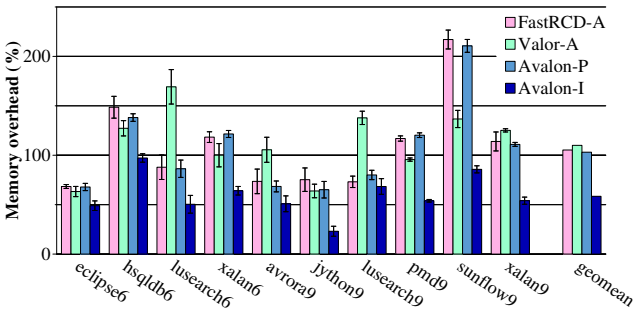


Figure 6. Space overhead of relevant implementations.

	JMM	SC	RS	RI
hsqldb6	Infinite loop	None	None	None
sunflow9	Null ptr exception	None	None	None
jbb2000	Corrupt output	Corrupt output	None	None
jbb2000	Infinite loop	None	None	None
sor	Infinite loop	None	None	None
lufact	Infinite loop	None	None	None
moldyn	Infinite loop	None	None	None
raytracer	Fails validation	Fails validation	None	None

Table 3. Erroneous behaviors possible under the Java memory model, and what errors are allowed by SC, RS, and RI. jbb2000 has two distinct errors, each caused by a different data race.

model than Avalon-I (RSx vs. RIx), Avalon-I does not have Valor-A's disadvantage of asynchronous exceptions and safety issues for unsafe languages (Section 4), and Avalon-I incurs significantly lower space overhead than other implementations (Section 8.6).

8.5 Scalability

The implementations that avoid exceptions by waiting at conflicts—FastRCD-A, Valor-A, Avalon-P, and Avalon-I—incur not only instrumentation overhead but also overhead from waiting. In an effort to isolate the cost of waiting, this section evaluates scalability across varying numbers of applications threads. Three of the evaluated programs support varying the number of application threads (Table 1). Figure 5 compares execution time of three pairs of waiting and non-waiting versions of otherwise-identical implementations. We omit Avalon-P since Avalon-I spends more time waiting at conflicts. We use *Avalon-I-no-wait* (mentioned in Section 8.3) as a comparison configuration for Avalon-I.

Overall, waiting at conflicts is not detrimental to scalability. For all three benchmarks, the waiting implementations (FastRCD-A, Valor-A, Avalon-I) scale as well as the non-waiting implementations (FastRCD, Valor, Avalon-I-no-wait).

8.6 Space Overhead

The implementations use memory to maintain write and read metadata. Figure 6 shows the space overhead of all implementations that wait at conflicts, relative to unmodified JVM execution. For each execution, we define its space usage as the maximum memory used after any full-heap garbage collection (GC). We omit luindex9 since its baseline execution triggers no full-heap GC.

On average, FastRCD-A adds 105% space overhead in order to maintain precise write and read metadata, which is particularly costly for variables with concurrent reader regions. Avalon-P maintains the same metadata and thus adds similar memory overhead (103% on average). Although Valor-A avoids storing per-variable read metadata, it adds high space overhead (110% on average) for per-thread read logs.

Avalon-I adds 58% average space overhead, about half as much as the other approaches. Avalon-I uses less space by maintaining imprecise reader information and by compressing per-variable metadata into a single word.

8.7 Evaluating RI's Behavior

Just as most programmers today do not reason about weak memory models, we expect them to be largely oblivious to an RIx-by-default memory model. After all, RI behavior is only possible when there is a data race—which is still an error. Instead, we are interested in what behaviors RI allows in practice for real data races.

Table 3 shows erroneous behaviors due to data races that are possible under Java's memory model (JMM) [61], and whether other memory models permit them, for the DaCapo programs, SPECjbb2000 [83], and the Java Grande benchmarks [82]. The *JMM*, *SC*, and *RS* columns are from prior work that exposes weak memory errors using *adversarial memory* [41, 76].[6] The RI column is based on our manual inspection of and reasoning about these errors. The *SC* column shows that 2 of 8 errors are actually possible under SC. In contrast, these erroneous behaviors cannot be exposed under RS or RI, according to our manual inspection of, and reasoning about, these errors. We find that RI avoids these errors because they generally involve violations of visibility or atomicity—violations that RI mostly avoids, since it provides write atomicity and read isolation, violating serializability only in case of write skew (Section 5). This (admittedly limited) empirical study suggests that RI tolerates erroneous behaviors in practice, allowing fewer erroneous behaviors than even SC.

[6] Despite avrora9's many dynamic data races (Table 1), prior work that uses adversarial memory has mostly failed to expose erroneous behavior in avrora9 [41, 76]. However, by using hundreds of trials, Cao et al. found that adversarial memory can occasionally expose errors in avrora9 [28].

9. Discussion

This paper shows how to improve availability under RSx, and it introduces RIx and Avalon to increase availability beyond RSx-based approaches. These approaches advance the state of the art but incur run-time costs that are likely to be too high for most production settings (an issue also faced by software approaches that provide RSx [14]). In Section 8.3, we find that virtually all overhead comes from detecting memory access conflicts. In contrast, existing work on custom *hardware* can efficiently detect memory access conflicts, in order to enforce RSx [15, 59] or other region-serializability-based consistency model [7, 45, 62, 80].

We believe that future work can apply our work's insights and approaches to hardware and achieve similar benefits. First, future work can extend architecture support for RSx to wait at detected conflicts, either (a) avoiding conflicts or (b) incurring deadlock and generating a consistency exception. Second, future hardware support can target RIx, improving availability and enabling simpler, more efficient designs since conflict detection does not need to be precise.

10. Related Work

To the best of our knowledge, prior work has not considered or addressed the problem of availability for memory consistency models that provide fail-stop semantics. This section compares our work against prior work not already covered in Section 2.

Serializability of bounded regions. Prior work supports a memory model based on serializability of regions *smaller* than SFRs [7, 60, 62, 76, 80]. Besides being weaker than RSx, bounded region serializability requires restricting compiler optimizations across region boundaries. Our RIx model and Avalon analyses relax RSx in a different way: they retain full SFRs but provide isolation without serializability, in order to improve availability and reduce costs.

Detecting and tolerating data races. Sound and precise data race detectors can provide RSx by throwing a consistency exception on every detected data race [35, 92]. However, state-of-the-art data race detectors slow programs by an order of magnitude or rely on custom hardware [33, 40].

Clean is a data race detector that detects write–write and write–read races but not read–write races [74]. By providing fail-stop semantics at the detected data races, Clean eliminates the most egregious weak memory model behaviors (so-called "out-of-thin-air" violations [22, 61]). Clean and Avalon both relax the requirement of detecting read–write conflicts precisely. While Clean avoids some erroneous behaviors, it does not provide RIx or any guarantee resembling region isolation.

By providing well-defined behavior for data races, Avalon narrows the set of possible thread interleavings and behaviors, which is conceptually related to dynamic approaches that automatically avoid concurrency errors [46, 94, 95].

Avoiding deadlocks. Wang et al. avoid existing deadlocks in lock-based programs by avoiding potentially unsafe interleavings [89, 90]. In contrast, our approaches intentionally *risk* deadlocks for mutually dependent data races, essentially converting (cycles of) conflicts into deadlocks.

Deterministic execution. Systems that ensure deterministic multithreaded execution have employed mechanisms that are related to those used by FastRCD-A, Valor-A, and Avalon. *DMP* delays each region's writes until a point where all regions perform writes at the same time, in order to produce a deterministic outcome [12, 32]. *Dthreads* exploits existing relaxed memory models in order to perform loads and stores in isolation and merge them at synchronization operations [57]. In contrast, Avalon detects conflicts in order to provide the RIx memory model, and it waits at conflicts in an effort to increase availability.

Region isolation in other contexts. Database management systems commonly support *snapshot isolation* (SI) as an alternative to strict serializability semantics for transactions [36, 39, 70]. These systems typically implement SI using *multi-versioning* to track multiple versions of data, based on a globally ordered timestamp that provides a total order for all committed transactions. Maintaining globally ordered transactions and multiple versions of data at the language level would incur high overhead and poor scalability.

Some software transactional memory (STM) systems use SI as the isolation model [50, 55]. In contrast, our work focuses on SI-based semantics for memory consistency models. Furthermore, databases and STMs execute transactions speculatively (i.e., conflicts lead to rollbacks), whereas our work converts data races with ill-defined semantics to well-defined behaviors, and employs SI in an effort to increase availability and reduce costs and complexity.

Burckhardt et al. consider the effects of isolation on program regions [25]. Their work introduces a new task-parallelism-based programming model, while we focus on enforcing the RIx memory model for existing programs written in legacy languages. Their work requires programmers to choose isolation types and specify how to merge on conflict, whereas our work is fully automatic.

11. Conclusion

Programming languages and runtime systems need a precise and practical memory model definition that balances semantics, performance, and availability. This paper is the first to address the challenge of availability for strong memory consistency models that throw consistency exceptions, which is crucial for their practical adoption.

We introduce FastRCD-A, Valor-A, the RIx memory model, and Avalon, which significantly reduce consistency exceptions compared with prior approaches. Overall, our approaches generate significantly fewer consistency exceptions than prior approaches, while providing competitive performance and scalability. Furthermore, RIx increases availability and enables efficient designs, without exposing erroneous behaviors, according to a study of real errors. Ultimately, our approaches advance the state of the art by providing new and compelling design points in the availability–performance–semantics tradeoff space.

Acknowledgments

We thank Brandon Lucia, Rui Zhang, Man Cao, Jake Roemer, and Aritra Sengupta for helpful discussions and the anonymous reviewers for 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 K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1996.

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

[5] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[6] A. Adya, B. Liskov, and P. O'Neil. Generalized Isolation Level Definitions. In *ICDE*, pages 67–78, 2000.

[7] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. BulkCompiler: High-Performance Sequential Con-

sistency through Cooperative Compiler and Hardware Support. In *MICRO*, pages 133–144, 2009.

[8] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[9] 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.

[10] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA*, pages 316–327, 2005.

[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*, pages 1–10, 1995.

[12] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, pages 53–64, 2010.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman, 1986.

[14] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.

[15] S. Biswas, R. Zhang, M. D. Bond, and B. Lucia. Efficient Architecture Support for Region-Serializability-Based Consistency. Technical Report OSU-CISRC-4/17-TR01, Computer Science & Engineering, Ohio State University, 2017.

[16] 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.

[17] S. M. Blackburn and K. S. McKinley. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *PLDI*, pages 22–32, 2008.

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

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

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

[21] H.-J. Boehm and S. V. Adve. You Don't Know Jack about Shared Variables or Memory Models. *CACM*, 55(2):48–54, 2012.

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

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

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

[25] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent Programming with Revisions and Isolation Types. In *OOPSLA*, pages 691–707, 2010.

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

[27] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable Isolation for Snapshot Databases. In *SIGMOD*, pages 729–738, 2008.

[28] M. Cao, J. Roemer, A. Sengupta, and M. D. Bond. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future. In *ISMM*, pages 99–110, 2016.

[29] M. Christiaens and K. De Bosschere. TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs. In *JVM*, pages 15–15, 2001.

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

[31] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, pages 67–78, 2010.

[32] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, pages 85–96, 2009.

[33] 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.

[34] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM Can Be More than a Research Toy. *CACM*, 54:70–77, 2011.

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

[36] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database Replication Using Generalized Snapshot Isolation. In *SRDS*, pages 73–84, 2005.

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

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

[39] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

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

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

[42] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-memory Multiprocessors. In *ASPLOS*, pages 245–257, 1991.

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

[44] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, pages 102–113, 2004.

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

[46] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated Concurrency-Bug Fixing. In *OSDI*, pages 221–236, 2012.

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

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

[49] E. Knapp. Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[50] I. Kuru, B. K. Ozkan, S. O. Mutluergil, S. Tasiran, T. Elmas, and E. Cohen. Verifying Programs under Snapshot Isolation and Similar Relaxed Consistency Models. In *TRANSACT*, 2014.

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

[52] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[53] C. Lin, V. Nagarajan, and R. Gupta. Efficient Sequential Consistency Using Conditional Fences. In *PACT*, pages 295–306, 2010.

[54] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS*, pages 273–286, 2012.

[55] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson. SI-TM: Reducing Transactional Memory Abort Rates Through Snapshot Isolation. In *ASPLOS*, pages 383–398, 2014.

[56] H. Litz, R. J. Dias, and D. R. Cheriton. Efficient Correction of Anomalies in Snapshot Isolation Transactions. *TACO*, 11(4):65:1–65:24, 2015.

[57] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *SOSP*, pages 327–336, 2011.

[58] 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.

[59] 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.

[60] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, pages 277–288, 2008.

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

[62] 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.

[63] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-Preserving Compiler. In *PLDI*, pages 199–210, 2011.

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

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

[66] 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.

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

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

[69] PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions, 2012. `http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[70] D. R. K. Ports and K. Grittner. Serializable Snapshot Isolation in PostgreSQL. *VLDB*, 5(12):1850–1861, 2012.

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

[72] P. Ranganathan, V. Pai, and S. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *SPAA*, pages 199–210, 1997.

[73] 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.

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

[75] K. Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, pages 11–21, 2008.

[76] 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.

[77] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. In *WBIA*, pages 62–71, 2009.

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

[79] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *TOPLAS*, 10(2):282–312, 1988.

[80] 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.

[81] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-End Sequential Consistency. In *ISCA*, pages 524–535, 2012.

[82] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC*, pages 8–8, 2001.

[83] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01, 2001.

[84] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*, pages 2–13, 2005.

[85] 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.

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

[87] 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.

[88] L. Wang and S. D. Stoller. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In *PPoPP*, pages 137–146, 2006.

[89] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *OSDI*, pages 281–294, 2008.

[90] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The Theory of Deadlock Avoidance via Discrete Control. In *POPL*, pages 252–263, 2009.

[91] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *SPAA*, pages 285–296, 2008.

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

[93] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *SPAA*, pages 265–274, 2008.

[94] J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *ISCA*, pages 325–336, 2009.

[95] J. Yu and S. Narayanasamy. Tolerating Concurrency Bugs Using Transactions as Lifeguards. In *MICRO*, pages 263–274, 2010.

[96] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*, pages 221–234, 2005.

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