All That Glitters Is Not Gold: Improving Availability and Practicality of Exception-Based Memory Models

Ohio State CSE technical report #OSU-CISRC-4/16-TR01, April 2016

Minjia Zhang
Ohio State University
zhanminj@cse.ohio-state.edu

Swarnendu Biswas
Ohio State University
biswass@cse.ohio-state.edu

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

Abstract

Modern shared-memory languages such as Java and C++ provide weak or undefined semantics for executions with data races. Existing work proposes stronger memory consistency models that ensure strong semantics but may throw a *consistency exception* in the presence of a data race. Even in well-debugged programs, consistency exceptions may occur unexpectedly, hurting availability.

This paper is the first to consider the problem of availability for memory models that throw data races. We introduce extensions to existing approaches that provide consistency based on *region serializability*; these extensions enable avoiding most consistency exceptions. To improve availability further, we introduce a new memory model called *SIx* based on *snapshot isolation* of code regions and a new approach called *Snappy* that provides SIx. We introduce two variants of Snappy that provide different performance–availability tradeoffs, while still ensuring SIx.

Our evaluation on real Java programs shows that our approaches provide new and compelling points in performance–availability tradeoff space. This work thus represents a promising direction for dealing with the key issue of availability for memory consistency based on fail-stop semantics.

1. Introduction

In order to achieve high performance, compilers and architectures for shared-memory parallel programs perform optimizations, such as eliminating and reordering memory accesses, assuming no dependences with concurrent threads. These optimizations are only restricted from reordering across synchronization operations (e.g., lock acquire and release). This approach allows shared-memory languages and hardware to provide strong guarantees—sequential consistency (SC), as well as serializability of code regions bounded by synchronization operations—for program executions that are free of data races, which are conflicting accesses to the same variable that are not ordered by synchronization operations. However, for executions with data races, compiler and hardware optimizations lead to erroneous, unpredictable, often undefined behavior [2, 17]. These guarantees are formalized in the *DRF0* memory consistency model [4], which ensures strong semantics for data-race-free executions only. Java and C++ and other shared-memory languages provide variants of DRF0 [19, 55].

To address this concern, researchers have proposed memory consistency models that provide strong semantics for all program executions. Notable among these are models that provide consistency based on *serializability of synchronization-free regions* (SFRs), which we call *region serializability* (RS), and use the terms region and SFR interchangeably. Under RS, an execution is equivalent to some execution in which SFRs (executed sequences of instructions that do not contain synchronization operations) appear to execute serially, i.e., without interruption by other threads [14, 53, 61]. RS is appealing because (1) it provides the same strong guarantees for all executions that DRF0 already provides but *only* for race-free executions, and (2) it does not restrict compiler and hardware optimizations, which already respect synchronization operations as region boundaries. However, enforcing RS seems inherently problematic due to supporting unbounded speculative execution. Researchers have thus introduced a memory consistency model that we call *RSx* that treats data races as errors, potentially throwing a *consistency exception* for a data race, but otherwise ensuring RS [14, 53]. (Furthermore, other approaches treat some or all data races as errors [31, 68, 80].)

Problem. Under RSx, data races are errors, just like buffer overflows and other memory errors in memory- and type-safe languages such as Java. In contrast, under DRF0, data races lead to undefined semantics, like buffer overflows in unsafe languages such as C++. In an RSx-by-default world, programmers will need to identify and eliminate most or all data races that cause consistency exceptions during testing, just as programmers now debug in response to exceptions from buffer overflows and other memory errors in safe languages. Nonetheless, even well-tested software may contain unknown data races that may manifest only under certain production environments, inputs, or thread interleavings [46, 63, 76], leading to unexpected consistency exceptions. Unexpected exceptions hurt the *availability* (this paper's term for exception freedom) of production systems.

Our approach. This paper is the first to our knowledge to consider and address the problem of availability in memory consistency models that generate consistency exceptions. We first introduce approaches that avoid consistency exceptions under RSx. We modify approaches from prior work, called *FastRCD* and *Valor*, that provide RSx [14]. Our approaches, called *FastRCD-A* and *Valor-A*, introduce *waiting* at program points that detect conflicts, in an effort to tolerate the conflict and avoid raising an exception. An evaluation on benchmarked versions of large, real Java programs shows that FastRCD-A and Valor-A improve availability substantially, compared with their counterparts that do not wait at conflicts. However, RSx is a strong model that may inherently limit availability and/or cost and complexity, whether implemented purely in software or with hardware support.

We thus introduce a new memory model called *SIx* based on *snapshot isolation* of regions (SI). While SI is weaker than serializability, it ensures region isolation and provides significantly stronger guarantees for racy executions than the weak or undefined semantics provided by DRF0.

To provide SIx, we introduce an approach called *Snappy* that correctly "tolerates" *read–write conflicts* by deferring handling of the conflicts until region end. Our evaluation shows that Snappy provides *even* better availability than FastRCD-A and Valor-A, although it incurs high overhead in order to detect conflicts precisely.

Our final insight is that Snappy can detect read–write conflicts *imprecisely* (false positives but no false negatives) without jeopardizing support for SIx. We leverage this idea to introduce a new approach, called *Snappy-I*, that represents variables' last reader information imprecisely, enabling lower run-time costs and complexity than the other approaches. However, Snappy-I's reduced precision gives up most of the availability gains that Snappy-P provides over the approaches that provide RSx.

Overall, our exploration and evaluation of the design space provides the following main conclusions. First, by waiting at conflicts, approaches that support RSx can increase availability substantially. Second, providing the SIx model enables higher availability than RSx. Third, relaxing precise conflict detection while providing SIx leads to significant performance benefits, but gives up most of the availability gains of SIx over RSx.

Although the overheads of these software-only approaches are too high for many production settings, future work could leverage custom hardware support to provide low-overhead RSx and SIx. We expect that supporting SIx instead of RSx could enable simpler hardware designs, just as Snappy-I yields a simpler software design than its competitors.

Contributions. To our knowledge, this paper is the first to consider the problem of *availability* for memory consistency models that throw consistency exceptions. The paper makes the following contributions:

- We introduce approaches for providing RSx that differ from prior work in that they wait at conflicts in order to avoid exceptions.
- We propose a new memory consistency model called *SIx* that is weaker than RSx but still ensures isolation of regions for racy executions.
- We present *Snappy*, an approach that enforces SIx. We introduce two designs with differing tradeoffs: *Snappy-P* uses full precision, while *Snappy-I* tracks reads imprecisely, trading availability for performance.
- We evaluate and compare availability, run-time performance, scalability, and other characteristics of our approaches and existing approaches that provide RSx and SIx. Our results show that our approaches provide new and compelling points in the performance–tradeoff space.

2. Background and Motivation

This section motivates the benefits of memory consistency based on serializability of code regions. It then describes two key challenges with consistency based on region serializability: (1) availability and (2) run-time costs and complexity.

2.1 Memory Consistency Models

Modern shared-memory languages such as Java and C++ provide variants of the *DFR0* memory model, introduced by Adve and Hill in 1990 [4, 19, 55]. DRF0 (and its variants) provide a strong guarantee for well-synchronized, or *data-race-free*, executions: *serializability of synchronization-free regions* (SFRs) [2, 53].[1] An execution has a data race if two accesses conflict (accesses to the same variable and at least one is a write) and are not ordered by the happens-before relation (the union of program and synchronization order) [44]. An SFR is a dynamic sequence of executed instructions bounded by synchronization operations (e.g., lock acquires and releases) with no intervening synchronization operations. An execution is region serializable if it is equivalent to some serial execution of regions (i.e., some global order of non-interleaved regions).

However, for executions with data races, DRF0 provides weak or no behavior guarantees [2, 17, 18, 20, 21]. C++'s memory model gives undefined semantics for data races [19]. The Java memory model (JMM), on the other hand, provides well-defined but weak semantics for racy executions, in an effort to preserve memory and type safety [55]. However, as researchers later discovered, the JMM precludes common Java virtual machine (JVM) compiler optimizations [71]. The state of practice is that JVMs perform optimizations that violate the JMM [21]. According to Adve and Boehm, "The inability to define reasonable semantics for programs with data races is not just a theoretical shortcoming, but a fundamental hole in the foundation of our languages and systems" [2].

[1] DRF0 also provides *sequential consistency* (SC) [45] for DRF0 executions. SFR serializability implies SC.

Despite much effort by researchers and practitioners, data races are difficult to avoid, detect, fix, and eliminate (e.g., [1, 14, 22, 23, 25, 26, 31, 33, 34, 36, 41, 58–60, 65, 67, 70, 77, 78, 82]). Data races often manifest only under certain environments, inputs, and thread interleavings, allowing them to go undetected [38, 52, 76, 83]. Data races thus occur unexpectedly in production systems, sometimes with severe consequences [46, 63, 76]. They often indicate concurrency bugs such as atomicity, order, and sequential consistency violations [52]. Thus, systems that execute with weak or undefined semantics due to data races are not just a problem in theory but in practice.

In spite of the shortcomings of DRF0-based memory models, languages and systems continue to use them in order to maximize performance. DRF0 allows compilers and hardware to perform uninhibited *intra-thread* optimizations, as long as optimizations do not cross synchronization operations. Any attempt to provide stronger consistency must consider the impact of restricting optimizations.

Sequential consistency. Much work has focused on providing *sequential consistency* (SC)[2] as the memory consistency model [2, 3, 37, 47, 48, 57, 66, 72, 74, 75]. Enforcing *end-to-end* SC (i.e., SC with respect to the original program) requires restricting optimizations by both the compiler and hardware. (In contrast, providing SC in the compiler or hardware alone does not provide end-to-end SC.)

Although SC is certainly stronger than the undefined or weak behaviors that DRF0 provides for racy executions, it is not a particularly strong model. Programmers tend to think in terms of operations that are larger than individual memory accesses, expecting a multi-access operation such as `x++` or `buffer[index++] = 42` to execute atomically (regardless of the actual memory model). Adve and Boehm argue that "programmers do not reason about correctness of parallel code in terms of interleavings of individual memory accesses" and that SC "does not prevent common sources of concurrency bugs ..." [2].

Region serializability. An alternative to DRF0 and SC is memory consistency based on *region serializability*. Notably, region serializability of SFRs, which we abbreviate as *RS*, extends the same guarantees to *all* executions that DRF0 provides for race-free executions only. Furthermore, approaches that provide RS can generally allow uninhibited compiler and hardware optimizations, which already do not cross synchronization boundaries.

Since enforcing RS or detecting all data races are both expensive (Section 9), prior work provides a memory model that either (1) ensures RS or (2) generates a *consistency exception*—but only if there exists a data race. This paper refers to this memory model as *RSx*. RSx allows for some flexibility: an approach does *not* need to incur the cost and complexity of soundly and precisely detecting all data races nor all RS violations. Prior work typically provides RSx by checking for region conflicts [14, 53]. A *region conflict* occurs when one region's access conflicts with an accessed performed by another ongoing region.[3]

Although RSx provides strong, well-defined semantics, supporting it incurs a major disadvantage: the possibility of consistency exceptions. Another challenge is the cost and complexity of detecting region conflicts.

2.2 Drawbacks of Providing RSx

Availability. RSx provides well-defined semantics even for racy executions. On the other hand, any racy execution can throw a consistency exception, essentially trading availability for strong, well-defined semantics. Data races occur unexpectedly, and then may or may not manifest unexpectedly as consistency exceptions depending on whether the race manifests as a region conflict.

Under RSx, data races are potential fail-stop errors, just like buffer overflows and null pointer exceptions in memory- and type-safe languages such as Java. Ideally, programmers would eliminate nearly all data races by addressing consistency exceptions encountered during testing and early production runs (e.g., alpha and beta testing). An additional mitigating factor is that programmers or automatic techniques may be able to *handle* consistency exceptions in a way that preserves availability. Nonetheless, we expect that consistency exceptions will occur unexpectedly and affect availability—just like null pointer exceptions—which may frustrate developers and users *more than* the (often silent and unknown) consequences of racy executions under DRF0. Prior work on memory models that generate consistency exceptions has not considered the issue of availability nor how to reduce exceptions [14, 31, 53, 56, 73].

Performance. Existing work that provides RSx incurs significant cost and complexity, whether implemented in hardware or software. *Conflict Exceptions* augments the cache coherence protocol in order to detect region conflicts, and adds on-chip network traffic and space overheads, making cache evictions and region boundaries more expensive [53]. Biswas et al. present two software-only approaches for providing RSx [14]. The first, *FastRCD*, slows down executions by 3.7X on average. The second approach, *Valor*, slows executions by 2.0X on average, through lazy detection of read–write conflicts. While advantageous for performance, it cannot tolerate read–write conflicts leading to consistency exceptions, although we find this problem is not substantial in practice on average. Valor incurs two other disadvantages: it provides imprecise exceptions and cannot easily handle unsafe languages such as C++ (Section 4.2). Common to existing software and hardware approaches for providing RSx is the cost and complexity of tracking the last region(s) to *read* each variable (for example, Valor has to log reads [14]), in order to detect or infer read–write conflicts accurately.

[2] Under SC, operations appear to interleave in some order that conforms to program order [45].

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

3. Goals and Overview

This paper's goals are to explore alternatives to current approaches that provide RSx and to develop approaches that provide better performance–availability tradeoffs than existing approaches. Our evaluation shows that current approaches have inherent limitations. We explore not only approaches that provide RSx but also a memory model that is weaker than RSx but is still *principled*, meaning that it provides well-defined semantics at region granularity, and does not inhibit compiler and hardware optimizations.

Section 4 develops mechanisms for avoiding consistency exceptions under RSx, which improve availability significantly. In an effort to improve availability further, Section 5 introduces a memory model called SIx. Section 6 presents two approaches for providing SIx that represent different points in the performance–availability space.

4. Increasing Availability Under RSx

This section extends two analyses from prior work [14] that provide RSx, *FastRCD* and *Valor*, in order to avoid consistency exceptions while still providing RSx. While these extensions by themselves are not huge intellectual contributions, our work is the first to introduce and evaluate them.

4.1 FastRCD and FastRCD-A

FastRCD is an analysis from prior work that provides RSx. Our presentation of FastRCD, including notations and algorithms, is closely based on prior work's presentation [14]. We extend FastRCD to *wait*, instead of throwing consistency exception, when it detects a region conflict. We call the resulting analysis *FastRCD-A* (Available).

FastRCD uses *epoch optimizations* from an existing data race detection analysis called *FastTrack* [36]. The analysis maintains, for each shared variable, (1) the last region that wrote the variable and (2) the last region(s) (one per thread) that read the variable since the last write. The analysis identifies regions by maintaining a per-thread logical clock for each thread; a thread's clock starts at 1, and the analysis increments it every time the thread executes a region boundary. FastRCD uses 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* that maps each thread to a clock value (or 0 if not present in the map).

Our FastRCD-A analysis extends FastRCD by *waiting* at detected region conflicts, until either (1) the region conflict no longer exists, in which case the analysis proceeds, or (2) the analysis detects a cyclic waiting dependency, in which case the analysis throws a consistency exception. Algorithms 1, 2, and 3, show FastRCD-A's analysis at region boundaries and program writes and reads, respectively.

Algorithm 1 REGION BOUNDARY [FastRCD-A]: thread T's region ends

1: incClock(T) ▷*Increment value returned by clock(T)*

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

1: **let** c@t $\leftarrow \mathcal{W}_x$
2: **if** c@t $\neq$ epoch(T) **then** ▷*First write to x by this region?*
3:     **if** t $\neq$ T $\wedge$ 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' $\mapsto$ c' **in** $\mathcal{R}_x$ **do**
9:         **if** t' $\neq$ T $\wedge$ clock(t') $=$ c' **then** ▷*Read–write conflict?*
10:             **if** deadlocked **then**
11:                 **throw** consistency exception
12:             **else**
13:                 Retry from line 1
14:     $\mathcal{W}_x \leftarrow$ epoch(T) ▷*Update write metadata*
15:     $\mathcal{R}_x \leftarrow \emptyset$ ▷*Clear read metadata*

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

1: **if** clock(T) $\neq \mathcal{R}_x$[T] **then** ▷*First read to x by this region?*
2:     **let** c@t $\leftarrow \mathcal{W}_x$
3:     **if** t $\neq$ T $\wedge$ clock(t) $=$ c **then** ▷*Write–read conflict?*
4:         **if** deadlocked **then**
5:             **throw** consistency exception
6:         **else**
7:             Retry from line 1
8:     $\mathcal{R}_x$[T] $\leftarrow$ clock(T) ▷*Update read metadata*

Next, we describe the high-level operation of FastRCD-A's (and FastRCD's) analysis at program reads and writes. The first **if** statement in Algorithms 2 and 3 checks whether this region has already written or read this variable, respectively, in which case the algorithm needs not check for conflicts or update read/write metadata. Otherwise, the analysis checks for write–write and then read–write conflicts (Algorithm 2) or write–read conflicts (Algorithm 3) by checking whether the last writer and reader regions are still ongoing (i.e., clock(t) $=$ c). Note that checking for read–write conflicts involves checking for conflicts with every other thread's "last reader" region of x. If a conflict is detected, FastRCD-A tries to tolerate the region conflict by letting T wait (i.e., retry from line 1). After checking for conflicts, the analysis at a write or read updates the variable's write or read metadata, respectively. Additionally, the analysis at a write clears the read metadata. Instrumentation atomicity needs to be guaranteed at these operations, as discussed in Section 7.

FastRCD differs from FastRCD-A in how it handles conflicts (lines 7 and 13 in Algorithm 2 and line 7 in Algo-

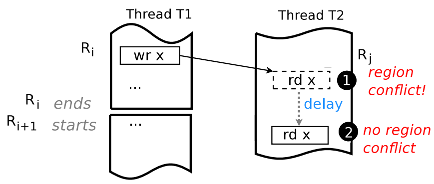


Figure 1. FastRCD-A avoids/tolerates some region conflicts: the read at (1) will cause a region conflict, but by waiting until (2), it succeeds without a consistency exception.

rithm 3). Instead of checking for a deadlock, FastRCD simply throws a consistency exception.

Figure 1 shows an example of how FastRCD-A works. The boxes represent program regions in each thread; time flows downward. Suppose that thread T2 has not accessed x in its current region before reading x at time (1). When T2 tries to read x at (1), it causes 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.

Waiting-induced deadlocks. Due to waiting, FastRCD-A can run into *waiting-induced deadlocks*. FastRCD-A maintains a global *region wait-for graph* in order to detect these deadlocks. The graph's nodes represent regions labeled with an epoch c@t, and the graph contains at most one region per node. Each node has at most one wait-for edge from and to another region. When one thread needs to wait for another thread to finish, a wait-for edge is added into the wait-for graph. An edge-chasing deadlock detection algorithm is used to find cycles in the graph [42].

Precise exceptions. FastRCD-A (and FastRCD) provide *precise* consistency exceptions, meaning that it suspends a thread's execution immediately before it performs a conflicting access. Precise exceptions are easier to handle and debug than imprecise exceptions [14].

4.2 Valor and Valor-A

As prior work and our evaluation show, FastRCD adds high run-time overhead, which is largely due to the cost of tracking last-reader metadata $\mathcal{R}_x$. Biswas et al. introduce an analysis called *Valor* that 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 extend Valor to *wait* at detected conflicts instead of throwing a consistency exception. We call the resulting analysis *Valor-A* (Available). 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 due to waiting on conflicts. However, unlike FastRCD-A, Valor-A cannot wait when it infers a read–write conflict: both the read and write accesses have already executed, so it is too late to try to avoid the conflict.

Appendix A presents the analysis for *Valor-A*, which is not pertinent to understanding the rest of this paper. Valor-A (and Valor) therefore throws *imprecise* consistency exceptions for read–write conflicts. Imprecise exceptions can be problematic because the reader region has already executed using inconsistent values. This problem is particularly acute in the context of a memory- and type-unsafe language such as C++, where so-called "zombie" regions can lead to corruption that would not be possible in any RS execution [14, 27, 40]. Furthermore, imprecise exceptions are difficult to use during development and debugging.

Thus, FastRCD is amenable to avoiding consistency exceptions, but it adds high run-time overhead. Valor improves performance by inferring read–write conflicts lazily, but it is not as well suited to avoiding consistency exceptions.

5. SIx: A New Strong Memory Model

The previous section presented our extensions to analyses that provide RSx, in an effort to avoid consistency exceptions as much as possible. Our evaluation shows that FastRCD-A and Valor-A generate significantly fewer exceptions than their non-waiting counterparts.

Can an approach provide availability comparable to or better than FastRCD-A's and performance comparable to Valor-A's? To achieve this goal, we *relax* consistency guarantees while still providing strong, principled semantics. This section introduces a new memory consistency model called SIx, and the next section introduces an analysis for supporting SIx.

This new SIx memory model is based on providing *snapshot isolation* of regions, or *SI*. SI is weaker than region serializability (RS): under SI, a region that reads a variable can be concurrent with a region that later writes the variable. Although SI is weaker than RS, prior work (in database systems) has observed that behaviors that lead to SI instead of RS semantics are rare in practice [35, 50]. Our insight here is that *an approach that provides SI can potentially have lower run-time costs and fewer conflicts because it does not need to detect read–write conflicts accurately*.

5.1 Snapshot Isolation of Regions

Prior work first introduced SI in the context of database processing [11]. Most of the database literature defines SI *operationally*. Here we give a definition based on execution equivalence, followed by a definition of *conflict SI*, which is a sufficient condition for SI. Both definitions are based closely on prior work.

Snapshot isolation. An execution is SI if it is equivalent to (has the same outcome as) some *serial* execution in which (1) each region R's reads[4] and writes *each* execute together as a group (without intervening accesses), (2) each region R's reads execute before its writes, and (3) another region Q's writes execute between R's reads and writes only if R and Q write distinct sets of variables [43].

[4] These reads do *not* include reads from variables that the region has already written. These "local" reads always read the value that the region wrote.

T1	T2
int t = x + y + 1; (1)	int t = x + y + 1; (1)
y = t;	x = t;

(a) SI and SC but *not* RS.

T1	T2
int t = x + 1; (1)	int t = x + 1; (1)
x = t;	x = t;

(b) SC but *not* SI (and thus not RS).

T1	T2
x = 1;	y = 1;
int t = y; (0)	int t = x; (0)

(c) SI but *not* SC (and thus not RS).

Figure 2. Example executions comparing SI 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.

In contrast, *serializability* demands equivalence to a serial execution in which all of a region's accesses (reads and writes) execute together without interruption [61, 62].

Figure 2 shows a few examples to help understand the difference between RS and SI as well as sequential consistency (SC). Figure 2(a) shows by example that SI is weaker than RS. In contrast, Figure 2(b)'s execution violates SI (and thus RS): under SI, the regions cannot be concurrent because their write sets overlap. SI provides isolation, but not necessarily sequential consistency (SC), as Figure 2(c) shows. Despite not subsuming SC, SI is likely to be more intuitive (in addition to more practical to enforce) than SC: programmers already reason about code regions, and, while SI provides isolation of regions, SC subjects programmers to subtle reasoning about interleavings of memory accesses (Section 2).

Conflict SI. SI, as defined above, is not directly useful for designing approaches that provide SI. Instead, we use a slightly stronger property, *conflict SI*, a sufficient condition for SI that a dynamic analysis can check on the fly. Conflict SI is analogous to *conflict serializability* [13, 79].

The following definitions, which lead up to conflict SI, are closely based on prior work [5, 6]. Adya's dissertation proves that conflict SI is a sufficient condition for SI [5].

A multi-threading execution consists of reads and writes executing in regions. The following notation describes read and write operations in an execution:

- $w_i(x_i)$: 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_j ($i = j$ is allowed)

The following definition captures the notion of ordering in a multi-threading execution:

Definition 1 (Time-precedes order)**.** *This 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 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.*

Definition 2 (Conflict SI)**.** *An execution is conflict SI if the following conditions hold:*

1. *Two concurrent regions cannot modify the same variable. That is, for any two writes $w_i(x_i)$ and $w_j(x_j)$ 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 $i \neq j$:*[5]
   (a) $e_j \prec_t s_i$ and
   (b) for any $w_k(x_k)$ in the execution such that $j \neq k$, either $s_i \prec_t e_k$ or $e_k \prec_t s_j$.

In the above definition, changing $s_i \prec_t e_k$ (in part 2b) to $e_i \prec_t s_k$ yields the definition of *conflict serializability*.

5.2 A Memory Model Based on Snapshot Isolation

We introduce a new memory model called *SIx* based on snapshot isolation of regions (SI). Similar to RSx (Section 2.1), under SIx, an execution either provides SI or throws a consistency exception—but only if the execution has a data race. In particular, SIx guarantees that each execution conforms to one of the following:

- a consistency exception, but only if the execution has a data race; or
- SI of regions if the execution does not throw a consistency exception.

If an execution is data race free, SIx inherently ensures not only SI but also RS. SIx is strictly stronger than DRF0, and it relaxes semantics from RS to SI only for racy regions, which have undefined or weak semantics under DRF0.

6. Snappy: Runtime Support for SIx

This section introduces *Snappy*, a novel, software-only approach that provides the SIx memory consistency model. Like FastRCD-A and Valor-A, Snappy detects and waits at write–write and write–read conflicts. Snappy differs from FastRCD-A and Valor-A—and prior work in general—in two major ways. First, its support for SIx enables "tolerating" read–write conflicts by deferring waiting to the end of the writer region. Second, as a result of its handling of read–write conflicts, Snappy can detect read–write conflicts imprecisely, enabling less costly tracking of last reader(s), while still preserving SIx (no deadlock without a data race).

6.1 Tolerating Read–Write Conflicts

During a region's execution, Snappy tracks each shared variable's last writer and last reader(s) accurately (just like FastRCD-A). Last-writer tracking allows Snappy to detect write–write and write–read conflicts precisely. Snappy handles read–write conflicts differently from both FastRCD-A and Valor-A: Snappy defers waiting on these conflicts until the end of the executing region, which still preserves

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

Algorithm 4 REGION BOUNDARY [Snappy]: T's region ends

1: incClock(T) ▷*Last region done; not ready to start next region*
2: **for each** t $\mapsto$ c **in** T.waitMap **do**
3:     **while** clock(t) = c **do** ▷*Read–write conflict still exists?*
4:         deadlocked $\leftarrow$ checkIfDeadlocked()
5:         **if** deadlocked **then**
6:             **throw** consistency exception
7: T.waitMap $\leftarrow \emptyset$
8: incClock(T) ▷*Ready to start next region*

SIx. With this design, Snappy opens new avenues for the improvement of availability and performance of exception-based memory models.

To present Snappy, we reuse clock(T), epoch(T), $\mathcal{R}_x$, and $\mathcal{W}_x$ from Section 4. In addition, we add the following notation specific to Snappy:

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. Clock values for threads not mapped in T.waitMap are considered to be 0.

Region boundaries. To support waiting at region boundaries, Snappy uses 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 to tolerate read–write conflicts at a region boundary.

Algorithm 4 shows how Snappy 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 Snappy detects deadlock.

Writes and reads. Algorithms 5 and 6 show the analysis that Snappy performs at each program write and read, respectively. The analysis differs from the analysis for FastRCD-A (Algorithms 2 and 3) in the following ways. First, in Snappy, a thread T waits at write–write and write–read conflicts (line 3 in Algorithm 5 and line 3 in Algorithm 6) 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 waits until the writer thread t's clock is at least two greater than the variable's clock c, i.e., clock(t) $\geq$ c + 2.

Second, when T detects a read–write conflict (line 9 in Algorithm 5), instead of waiting, it records the conflicting thread t' and its current clock c' in T.waitMap. T.waitMap

Algorithm 5 WRITE [Snappy]: thread T writes x

1: **let** c@t $\leftarrow \mathcal{W}_x$
2: **if** c@t $\neq$ epoch(T) **then** ▷*First write to x by this region?*
3:     **if** t $\neq$ T $\wedge$ clock(t) $\leq$ 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' $\mapsto$ c' **in** $\mathcal{R}_x$ **do**
9:         **if** t' $\neq$ T $\wedge$ clock(t') = c' **then** ▷*Read–write conflict?*
10:             T.waitMap$[t'] \leftarrow c'$
11:     $\mathcal{W}_x \leftarrow$ epoch(T) ▷*Update write metadata*
12:     $\mathcal{R}_x \leftarrow \emptyset$ ▷*Clear read metadata*

Algorithm 6 READ [Snappy]: thread T reads x

1: **if** clock(T) $\neq \mathcal{R}_x$[T] **then** ▷*First read to x by this region?*
2:     **let** c@t $\leftarrow \mathcal{W}_x$
3:     **if** t $\neq$ T $\wedge$ clock(t) $\leq$ c + 1 **then** ▷*Write–read conflict?*
4:         **if** deadlocked **then**
5:             **throw** consistency exception
6:         **else**
7:             Retry from line 1
8:     $\mathcal{R}_x$[T] $\leftarrow$ clock(T) ▷*Update read metadata*

needs to maintain only the latest clock value for every other thread t' (multiple values are possible due to waiting on multiple threads), so the analysis updates T.waitMap only if the new value is greater than the old value.

Examples. Figure 3 shows examples of how Snappy enforces SIx. In the figure, concurrent regions access the shared variables x and y. The gray dashed lines along with the synchronization operations, 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 3(a), 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 3(b), a cyclic dependence exists, but each region can reach its region boundary, allowing each to proceed. In contrast, FastRCD-A deadlocks for this example. In Figure 3(c), Snappy 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 Detecting Read–Write Conflicts Imprecisely

Although tolerating read–write conflicts allows Snappy to avoid exceptions that FastRCD-A and Valor-A encounter, our evaluation shows that it incurs about the same (high) overhead as FastRCD-A in order to detect all conflicts accurately when they occur. In developing Valor, prior work shows that the main cost of FastRCD is in tracking the last reader(s) for each variable precisely to detect read–write conflicts accurately [14].

This section proposes an alternate design for Snappy that focuses on *the approximation of the precise last reader(s)*

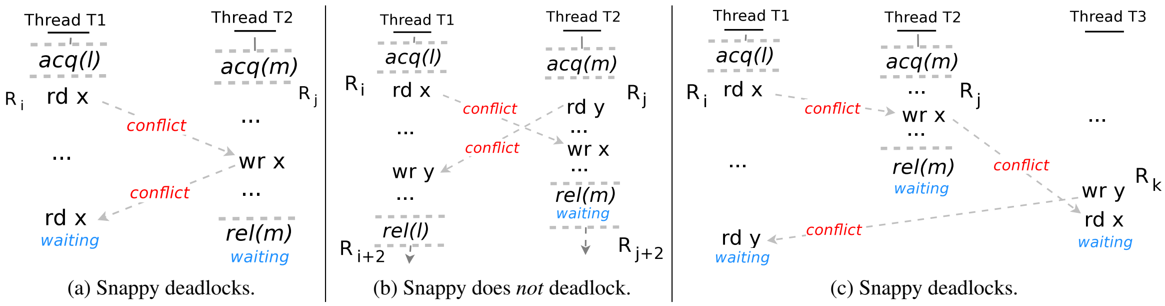


Figure 3. Examples showing how Snappy works. Dashed lines indicate where Snappy increments its clock. The exact synchronization operations (e.g., acq(m) versus rel(l)) are arbitrary and not pertinent to the examples.

to improve performance. For clarity, the rest of the paper refers to the design of Snappy described in Section 6.1 as *Snappy-P* (Precise) and the alternate design introduced here as *Snappy-I* (Imprecise).

The key insight of having imprecise readers is to let a conflicting write conservatively wait on potential read–write conflict(s) if the write cannot infer accurate read–write conflict(s). Importantly, Snappy-I still provides SIx: although Snappy-I may wait on a false read–write conflict, any deadlock must include at least one (true) write–read or write–write conflict (this fact holds for Snappy-P as well as Snappy-I), indicating a data race, as Appendix B shows.

In the design of *Snappy-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 needs to be inflated into a read map when there are multiple concurrent reader regions. Since Snappy-I does not require precise detection of read–write conflicts, it allows for a simpler and more efficient design.

Metadata representation. Like Snappy-P, Snappy-I maintains the epoch of the last writer for each shared variable x. However, it maintains precise last reader information only if there exists a single reader; for multiple readers, it does not maintain any information about them (so any ongoing region is a potential reader). As a result, Snappy-I represents a variable's last writer and reader metadata into a single unit of metadata (e.g., a single metadata word). This metadata has one of the following values:

$\mathsf{WrEx}_{c@t}$: x was last accessed by region c@t, and that region performed a write to x.

$\mathsf{RdEx}_{c@t}$: x was last accessed by region c@t, and that region performed only reads to x.

RdSh : At some point, there were multiple concurrent reader regions. Any ongoing region may have read x, but no ongoing region may have written x.

The first write in region c@t updates the variable's metadata to $\mathsf{WrEx}_{c@t}$. Similarly, the first read in c (if there is no prior write in the same region) updates the metadata to $\mathsf{RdEx}_{c@t}$. If a second read from a different thread reads the variable while the first read's region is still ongoing, Snappy-I promotes the metadata from $\mathsf{RdEx}_{c@t}$ to RdSh. Since all states can be encoded in a single metadata word, it is possible to use one *compare-and-swap* (CAS) instruction to update the metadata (Section 7).

Snappy-I's analysis. Algorithms 7 and 8 show the analysis that Snappy-I performs at each program write and read. In Algorithm 7, if the last write to x comes from the same region, the current write can skip the rest of the analysis operations (line 3) since the metadata does not need to be updated. If the last write is from the same thread T the write can update the metadata with the epoch of the current region (R_c). Otherwise, T handles $\mathsf{WrEx}_{*@t}$ and $\mathsf{RdEx}_{*@t}$ (* denotes "any clock value") as Snappy-P by detecting a write–write or a read–write conflict (lines 4–14). If the variable is in RdSh state, Snappy-I treats every other threads' ongoing region as having potential read–write conflicts with T's current write (lines 15–18).

In Algorithm 8, if the same region has already read or write the variable or the variable is in RdSh state, T does not need to update the metadata record (line 3). If the read is the first read in a region before any writes, the read overrides the metadata record from $\mathsf{WrEx}_{c'@t}$ to $\mathsf{RdEx}_{c@T}$, so that a write from a different thread can still detect a read–write conflict precisely at a $\mathsf{WrEx}_{c'@t}$ to $\mathsf{WrEx}_{c@T}$ transition instead of having a potential read–write conflict (an alternative is to change the metadata to RdSh, but it would lead to unnecessary imprecision and lower availability).

7. Implementation

We have implemented FastRCD-A, Valor-A, Snappy-P, and Snappy-I in Jikes RVM, a Java virtual machine [8, 9]. We choose Jikes RVM because (1) although it targets research, it performs competitively with commercial JVMs [14]; and (2) our implementations extend publicly available Jikes RVM implementations of FastRCD and Valor [14].

Following the FastRCD and Valor implementations, our implementations target IA-32 and extend both of Jikes RVM's just-in-time compilers to instrument synchronization operations and memory accesses. All implementations instrument the same points (field and array element accesses), demarcate regions in the same way (at lock, monitor, thread, and volatile operations), and reuse code as much as possible. The compilers instrument all application code, and the ap-

	Threads		Memory accesses		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).

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

1: **let** *oldMetadata* $\leftarrow \mathsf{x.state}$
2: **let** $\mathsf{c} \leftarrow \mathsf{clock(T)}$
3: **if** *oldMetadata* $\neq \mathsf{WrEx_{c@T}}$ **then** ▷*First write to* x *by this region?*
4: **if** *oldMetadata* $= \mathsf{WrEx_{*@t}}$ **then**
5: **let** $\mathsf{c@t} \leftarrow$ *oldMetadata*
6: **if** $\mathsf{clock(t)} \leq \mathsf{c} + 1$ **then** ▷*Write–write conflict?*
7: **if** deadlocked **then**
8: **throw** consistency exception
9: **else**
10: Retry from line 1
11: **else if** *oldMetadata* $= \mathsf{RdEx_{*@t}}$ **then**
12: **let** $\mathsf{c@t} \leftarrow$ *oldMetadata*
13: **if** $\mathsf{clock(t)} = \mathsf{c}$ **then** ▷*Potential read–write conflict?*
14: $\mathsf{T.waitMap}[t] \leftarrow \mathsf{clock(t)}$
15: **else if** *oldMetadata* $= \mathsf{RdSh}$ **then**
16: **for each** thread $\mathsf{t'}$ **do**
17: **if** $\mathsf{t'} \neq \mathsf{T}$ **then** ▷*Potential read–write conflicts?*
18: $\mathsf{T.waitMap}[t'] \leftarrow \mathsf{clock(t')}$
19: $\mathsf{x.state} \leftarrow \mathsf{WrEx_{c@T}}$

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

1: **let** *oldMetadata* $\leftarrow \mathsf{x.state}$
2: **let** $\mathsf{c} \leftarrow \mathsf{clock(T)}$
3: **if** *oldMetadata* $\neq \mathsf{RdEx_{c@T}} \wedge$ *oldMetadata* $\neq \mathsf{WrEx_{c@T}} \wedge$ *oldMetadata* $\neq \mathsf{RdSh}$ **then**
4: *newReadMetadata* $\leftarrow \mathsf{RdEx_{c@T}}$
5: **if** *oldMetadata* $= \mathsf{WrEx_{*@t}}$ **then**
6: **let** $\mathsf{c'@t} \leftarrow$ *oldMetadata*
7: **if** $\mathsf{T} \neq \mathsf{t} \wedge \mathsf{clock(t)} \leq \mathsf{c'} + 1$ **then** ▷*Write–read conflict?*
8: **if** deadlocked **then**
9: **throw** consistency exception
10: **else**
11: Retry from line 1
12: **else if** *oldMetadata* $= \mathsf{RdEx_{*@t}}$ **then**
13: **let** $\mathsf{c'@t} \leftarrow$ *oldMetadata*
14: **if** $\mathsf{clock(t)} = \mathsf{c'}$ **then** ▷*Concurrent reader?*
15: *newReadMetadata* $\leftarrow \mathsf{RdSh}$
16: $\mathsf{x.state} \leftarrow$ *newReadMetadata*

plication calls an instrumented compiled version of the Java libraries (the JVM, which is written in Java, calls a separately compiled, uninstrumented version of the libraries).

Following the FastRCD implementation, FastRCD-A and Snappy-P add two words of metadata for tracking writes and reads. While each variable has only a write epoch, its read metadata can be inflated to a pointer that points to a read map. In contrast, Snappy-I use one word of metadata: 21 bits for the clock, 9 bits for the thread ID, and 2 bits for encoding the state (write-exclusive vs. read-exclusive vs. read-shared). Snappy-I can reset clocks to either 0 or 1 at full-heap garbage collection to avoid overflow [14]; or it can ignore overflow, in which case false positive conflicts due to wraparound are unlikely. In all implementations, each per-field metadata is laid out beside the field, while an array's per-element metadata is referenced indirectly through the array's header.

Instrumentation atomicity. Following FastRCD, FastRCD-A, and Snappy-P guarantee instrumentation atomicity by "locking" one of the variable's metadata words (using an atomic operation and spin loop) and "unlocking" (using a store and fence) when updating the metadata. The instrumentation does not perform any synchronization when the instrumentation performs no metadata updates (for a read or write in the same region). These analyses require lock and unlock operations to ensure instrumentation atomicity, since metadata is two words or more (due to a possible read map). In contrast, following Valor, Valor-A and Snappy-I use a single word of metadata per variable, and ensure instrumentation atomicity by using a single atomic operation at the end of the analysis to atomically update the state.

Waiting at conflicts. Instrumentation for the FastRCD-A, Valor-A, Snappy-P and Snappy-I waits when it detects a region conflict. In order for a thread T to wait on another thread t's clock to change, T waits on a monitor associated with t; t notifies T that it has finished its region by broadcasting on the monitor at region boundaries.

8. Evaluation

This section evaluates the availability, performance, scalability, space usage, and other characteristics of our approaches and existing approaches for providing RSx and SIx.

8.1 Methodology

Benchmarks. We evaluate our implementations using benchmarked versions of large, real applications: the DaCapo benchmarks [15] with the default workload size, versions 2006-10-MR2 and 9.12-bach (distinguished with names suffixed by 6 and 9) [15]. We omit single-threaded programs and programs that Jikes RVM 3.1.3 cannot execute.

Experimental setup. For each implementation, we build a high-performance configuration of Jikes RVM that adaptively optimizes the code and uses the default, high-performance, generational garbage collector [16]. The garbage collector adjusts the heap size automatically at run time. Each performance result is the mean of 25 trials. We use special statistics-gather configurations to collect statistics, and report the mean of 10 trials. We report 95% confidence intervals for both statistics and execution times.

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

Run-time characteristics. Table 1 shows characteristics of the evaluated programs. The *Threads* columns report both threads created and maximum threads active at any time. The remaining columns show statistics collected with Snappy-P (the statistics from other configurations are similar, since these statistics are not specific to configurations). The *Memory accesses* are executed memory accesses (loads and stores of fields and array elements), which all configurations instrument. The *Conflicts* column shows how many conflicts of each type occur (during execution under Snappy-P). Conflicts vary significantly in count and type across programs, but they are generally many orders of magnitude smaller than total memory accesses, except for avrora9, which incurs millions of write–read conflicts.

The last two columns report executed synchronization-free regions (SFRs) and average memory accesses executed in each SFR. All programs except sunflow9 perform synchronization at least every 720 memory accesses.

8.2 Availability

Under the DRF0 consistency model, data races are essentially errors since they have undefined or weak semantics (Section 2.1). However, in practice, language and hardware implementations typically ignore data races silently (rather than treating them as fail-stop errors), and thus real programs—including those we evaluate—often contain data races that commonly manifest but do not cause problems under typical compilation and execution environments. In contrast, by providing RSx and SIx, our work follows a line of research that treats some or all data races as errors [14, 18, 24, 31, 53, 56, 73, 80]. In order to avoid frequent exceptions in production environments, developers would need to identify and fix data races that commonly lead to consistency exceptions when using Valor or Snappy or another technique that provides RSx or SIx (e.g., during in-house, alpha, and beta testing).

Since the programs we evaluate have *not* been developed or debugged under the assumption that data races are (fail-stop) errors, they throw consistency exceptions frequently under RSx and SIx. We compare the numbers of consistency exceptions generated under different approaches for RSx and SIx. From that, we extrapolate which approaches would be more likely to avoid exceptions *if programs were developed and debugged to avoid consistency exceptions*.

Table 2 compares consistency exceptions raised by FastRCD, Valor, FastRCD-A, Valor-A, Snappy-P, and Snappy-I. Our implementations do not actually generate exceptions; rather, they simply report the exception and allow execution to proceed. In case of a deadlock, the thread that created the cycle of dependences is allowed to proceed immediately. FastRCD and Valor report an exception whenever they detect a conflict. FastRCD-A, Valor-A, Snappy-P, and Snappy-I report an exception whenever they detect a deadlock. Valor and Valor-A also report an exception whenever they infer a conflict via a read validation failure.

Since some executed regions may have many related conflicts, the first row for each program is the number of dynamic regions that report at least one exception. The second row is the number of consistency exceptions reported during the program execution. Each result is the mean of exceptional regions or exceptions, $\pm$ 95% a confidence interval.

Waiting at conflicts. We first consider the effect of *waiting* at conflicts rather than generating an exception, i.e., Section 4's innovation that yields FastRCD-A and Valor-A. Based on comparing FastRCD-A with FastRCD and Valor-A with Valor, the effect of waiting at conflicts is substantial, reducing regions that report an exception by up to three orders of magnitude. Our evaluation of waiting at conflicts—which to our knowledge is the first such evaluation—suggests that this approach is generally effective at increasing the ability to avoid consistency exceptions while still preserving consistency models such as RSx and SIx.

RSx versus SIx. A key intended benefit of Snappy is that by providing SIx, it can potentially avoid consistency exceptions encountered by providing RSx. To evaluate this benefit, we compare primarily exceptions generated by Snappy-P versus FastRCD-A, which are identical except that Snappy-P can tolerate read–write conflicts. We find that Snappy-P generally avoids consistency exceptions compared to FastRCD-A: Snappy-P has fewer exceptional regions than FastRCD-A for 10 programs, never has more exceptional regions than FastRCD-A, and for the other 1 program (i.e., eclipse6) there is no statistically significant difference.

Relaxing Snappy's precision. Another potential benefit of Snappy is that by relaxing precision, it can improve performance. In order to evaluate the *cost* of this, we com-

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

Table 2. The number of consistency exceptions reported by approaches that provide RSx and SIx. For each program, the first row is dynamic regions that report at least one exception, and the second row is dynamic exceptions reported. Reported values are the mean of 10 trials, including 95% confidence intervals, rounded to two significant figures (if $\geq$1.0).

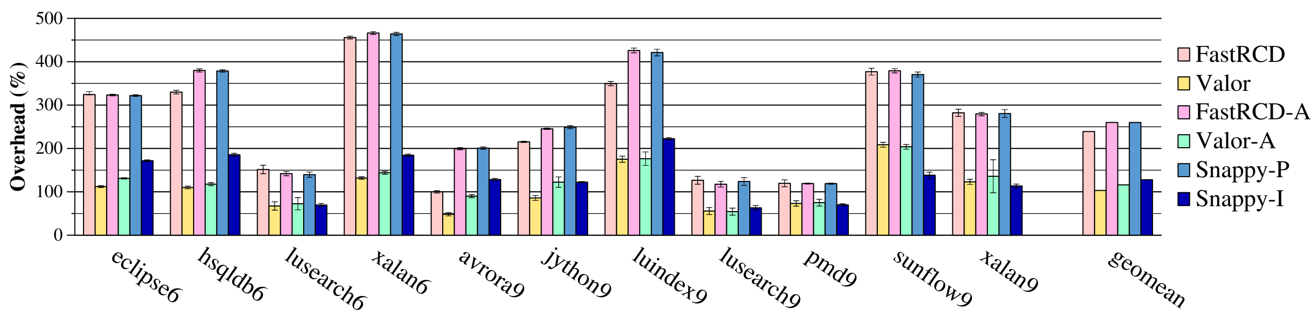


Figure 4. Runtime overhead added to unmodified Jikes RVM by FastRCD, Valor, and our implementations of FastRCD-A, Valor-A, Snappy-P, and Snappy-I.

pare reported consistency exceptions for Snappy-P and Snappy-I. Unsurprisingly, Snappy-I reports more exceptions than Snappy-P, since Snappy-I introduces waiting at region boundaries for spurious read–write conflicts. This effect is mixed across programs: for 6 programs Snappy-I generates more exceptions than Snappy-P, whereas we find no statistically significant difference for the other 5 programs.

Overall, waiting at conflicts and providing SIx instead of RSx both increase availability, while relaxing Snappy's precision decreases availability, but perhaps not excessively. The suitability of each approach not just on its availability but also on its performance, discussed next.

8.3 Performance

This section measures and compares the effect on performance of various approaches for providing strong memory models. Since executions are multithreaded, performance overheads include not just instrumentation overhead but also time spent on waiting, which differs among approaches due to different conditions for waiting. Section 8.5 attempts to separate out this cost by measuring performance for varied application thread counts.

Figure 4 shows the run-time overhead added by the configurations from Section 8.1 to execution on an unmodified JVM. As prior work shows, 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 additional overhead over FastRCD-A and Valor-A, respectively, due to the time spent waiting at conflicts.

Snappy-P tracks write and read metadata in the same way as FastRCD-A, so it unsurprisingly incurs similar overhead. However, Snappy-P incurs slightly less overhead than FastRCD-A because Snappy-P provides SIx and thus can relax its waiting at read–write conflicts (by deferring waiting until region end; Section 6). However, Snappy-P adds 260% overhead on average in order to provide precise conflict detection. In contrast, Snappy-I enables a significantly faster analysis that slows programs by 128% on average.

To understand the performance difference between Snappy-I and Snappy-P, we implemented a configuration (not shown in the figure) that tracks both read and write metadata in a way similar to Snappy-P (i.e., separate metadata words for the writer and reader(s), except that it does not track multiple readers precisely, instead using a simple "read shared" when multiple concurrent reader regions exist. On average this configuration incurs 75% of the overhead that Snappy-P incurs over Snappy-I, suggesting that most but not all of Snappy-I's performance advantage comes from its ability to represent its metadata in a single metadata word, leading to significantly simpler and cheaper instrumentation.

To isolate Snappy-I's overhead due to waiting at conflicts (versus instrumentation overhead), we also evaluated a Snappy-I configuration (not shown in the figure) that does not wait at conficts, but instead allows a thread to proceed immediately after detecting a region conflict. Compared with this "no waiting" configuration, the default Snappy-I configuration adds only 9.5% additional overhead (relative to baseline, unmodified execution), suggesting that little of Snappy-I's overhead is due to waiting at conflicts. This result makes sense conflicts are usually many orders of magnitude smaller than total memory accesses. The entire execution time therefore is still dominated by instrumentation overhead added to memory accesses.

8.4 Performance–Availability Tradeoff

In order to evaluate availability and performance together, we plot the previous availability and performance results in Figure 6. 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, $availability = \frac{\#\ memory\ accesses}{\#\ consistency\ exceptions + 1}$. (In contrast, taking the geomean of exceptions would be problematic because some values are 0.) Values closer to the top left corner represent a better performance–availability tradeoff.

Valor has the best performance, but its availability is significantly worse than FastRCD-A, Valor-A, Snappy-P, and Snappy-I. Snappy-P has the best availability, but its performance overhead is relatively high. Snappy-I and Valor-A arguably have the best tradeoff between availability and performance. We note that Snappy-I has lower space overhead than other implementations (Section 8.6), and it does not

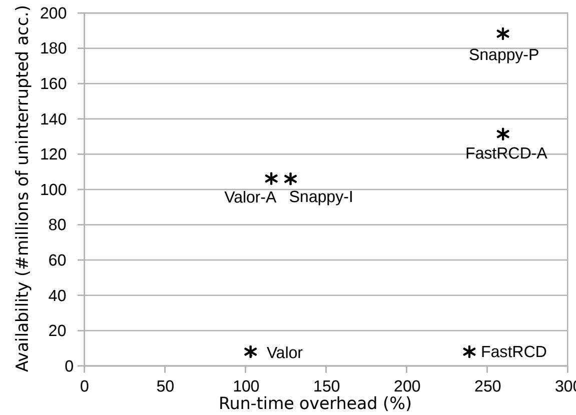


Figure 6. The comparison of performance and availability of different memory models.

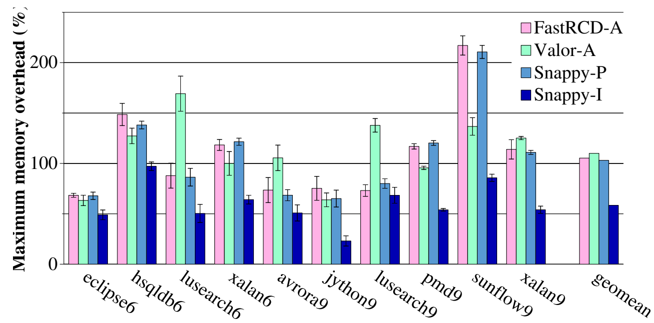


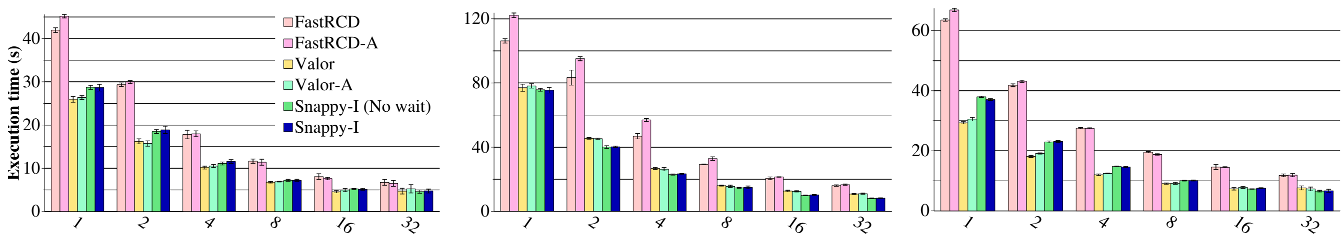
Figure 7. Run-time space overhead of FastRCD-A, Valor-A, Snappy-P, and Snappy-I.

have Valor-A's disadvantages of imprecise exceptions and safety issues for unsafe languages (Section 4.2).

8.5 Scalability

Approaches that avoid exceptions by waiting at conflicts—FastRCD-A, Valor-A, Snappy-P, and Snappy-I—incur not only instrumentation overhead but also overhead due to waiting. In an effort to separate out the conflicts, 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, for 1–32 application threads, the execution time of the waiting and non-waiting versions of implementations that are otherwise identical. We leave out Snappy-P and only show Snappy-I since the measurement here is whether waiting at conflicts hurt scalability instead of instrumentation overhead. We use Snappy-I (No wait) (introduced in the last subsection) as a comparison configuration to Snappy-I.

Overall, supporting RSx and SIx with waiting has no detrimental effect on scalability. For all three benchmarks, the waiting versions (FastRCD-A, Valor-A, Snappy-I) scale equally well as the non-waiting configurations (FastRCD, Valor, Snappy-I (No wait)).



(a) lusearch9 (b) sunflow9 (c) xalan9

Figure 5. Execution time of the configurations that can incur waiting versus configurations that do not incur waiting, for 1–32 application threads. The legend applies to all graphs.

8.6 Space Overhead

The approaches incur space overhead in order to represent write and read metadata. Figure 7 shows the space overhead of all configurations 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 GCs.

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. Snappy-P adds similar memory overhead (103% on average), which makes sense because it maintains the same metadata as FastRCD-A. Although Valor-A avoids storing per-variable read metadata, it still adds high space overhead (110% on average) due to maintaining per-thread read logs. Per-program space overheads depend largely on thread counts and region size (Table 1).

Snappy-I adds 58% average space overhead, about half as much as the other approaches. Snappy-I uses less space by not maintaining precise information about reads—particularly in the case of multiple readers—whereas the other configurations maintain precise information about reads (including Valor-A, which maintains them in pre-thread logs). Furthermore, Snappy-I is able to use a single metadata word per field and array element for write and read metadata.

9. Related Work

This section compares our work against prior work not already covered in Section 2.

Enforcing region serializability. Prior work has *enforced* serializability of synchronization-free regions (RS), relying on heavyweight support for speculative execution [61]. The costs and complexity for enforcing RS are similar to those encountered in software and (unbounded) hardware transactional memory implementations (e.g., [10, 28, 30, 39, 81]). Furthermore, operations such as as I/O and system calls are not generally amenable to speculative execution [40].

Serializability of bounded regions. Prior work supports a memory model based on serializability of regions *smaller* than SFRs [7, 54, 56, 69, 73]. This approach can enable architecture or analysis support that is less complex than for full SFRs. In addition to being weaker than RSx, bounded region serializability requires restricting compiler optimizations across region boundaries. Our SIx model and Snappy analyses relax RSx in a different way: they retain full SFRs but provide isolation but not atomicity, in an effort to improve availability and reduce costs and complexity.

Detecting and tolerating data races. Data race detectors that soundly and precisely check the happens-before relation [44] can provide RSx, by throwing a consistency exception on every detected data race. However, state-of-the-art happens-before detectors slow programs by nearly an order of magnitude on average or rely on custom hardware support [31, 36, 80]. (Although prior work reports slowdowns of only 2X for *Goldilocks* [31], Flanagan and Freund show that using realistic methodology would incur an estimated 25X average slowdown [36].

Recent work introduces a data race detector called *Clean* that detects write–write and write–read races but not read–write races [68]. By providing fail-stop semantics at the detected data races, Clean eliminates some of the most egregious weak memory model behaviors (e.g., so-called "out-of-thin-air" violations [21, 55]), Clean and Snappy both relax the requirement of detecting read–write conflicts precisely. However, Clean incurs high overhead in order to track the happens-before relation. It inherently cannot tolerate detected data races: waiting at an access can avoid a detected conflict but not a detected data race. Although Clean can avoid some erroneous behaviors, it does not provide SIx or any strong guarantee of isolation of regions.

Deterministic execution. Systems that provide deterministic multithreaded execution have employed mechanisms that are related to those used by FastRCD-A, Valor-A, and Snappy. *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, 29]. *Dthreads* exploits existing relaxed memory models in order to perform loads and stores in isolation and merge them at synchronization operations [51]. In contrast, Snappy detects conflicts in order to provide the SIx memory model, and it waits at conflicts in an effort to increase availability.

Snapshot isolation in other contexts. Database management systems routinely support SI instead of strict serializability semantics [32, 35, 64]. 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 programming language level would likely incur high overhead and poor scalability.

Prior work has used SI as the isolation model for software transactional memory (STM) systems [43, 49]. In contrast, our work focuses on SI-based semantics for memory consistency models. Furthermore, the database and STM work executes transactions speculatively (i.e., conflicts lead to rollbacks), while our work converts data races with ill-defined semantics to well-defined behaviors and employs SI instead of RS in an effort to increase availability and reduce costs and complexity.

10. Conclusion

The RSx memory model provides strong guarantees for all executions, not just data-race-free executions, but introduces two significant challenges in bringing RSx into practice: availability and performance. We address these challenges first by introducing extensions to existing approaches that provide RSx, which our evaluation shows significantly improved availability. We target even better availability by introducing the new SIx memory model and a novel approach for enforcing SIx called Snappy that enables configurations that have difference performance–availability tradeoffs. Our evaluation shows that Snappy-P reduces the chances of an unavoidable consistency exception significantly. A high-performance version of Snappy-I achieves much lower overhead, but in turn loses the availability benefit provided by SIx over RSx. Overall, this work represents an advance in the state of the art by introducing novel memory models and run-time approaches that enable new and compelling points in the performance–availability space for strong memory consistency models that use fail-stop semantics.

A. Valor-A Algorithms

In addition to the $\mathsf{epoch(T)}$ and $\mathsf{clock(T)}$ helper functions used by FastRCD, Valor uses the following notation (based closely on prior work [14]):

$\mathcal{W}_x$ – Represents last-writer metadata for variable x, as $\langle \mathsf{v}, \mathsf{c@t} \rangle$, where v is a *version* and $\mathsf{c@t}$ is an epoch. A variable's version starts at 0, and the analysis increments it at every write to the variable.

T.readLog – Represents thread T's read logs. Each entry in the log has the form $\langle \mathsf{x}, \mathsf{v} \rangle$, where x identifies the variable and v is x's version when it was read.

We extend Valor to *wait* at detected conflicts instead of throwing a consistency exception. We call the resulting analysis *Valor-A*. Algorithms 9–11 show Valor-A's analysis at writes, reads, and region boundaries.

Algorithm 9 WRITE [Valor-A]: thread T writes variable x

1: **let** $\langle \mathsf{v}, \mathsf{c@t} \rangle \leftarrow \mathcal{W}_x$
2: **if** $\mathsf{c@t} \neq \mathsf{epoch(T)}$ **then** ▷ *First write to x by this region?*
3:     **if** $\mathsf{t} \neq \mathsf{T} \wedge \mathsf{clock(t)} = \mathsf{c}$ **then** ▷ *Write–write conflict?*
4:         **if** deadlocked **then**
5:             **throw** consistency exception
6:         **else**
7:             Retry from line 1
8:     $\mathcal{W}_x \leftarrow \langle \mathsf{v{+}1}, \mathsf{epoch(T)} \rangle$ ▷ *Update write metadata*

Algorithm 10 READ [Valor-A]: thread T reads variable x

1: **let** $\langle \mathsf{v}, \mathsf{c@t} \rangle \leftarrow \mathcal{W}_x$
2: **if** $\mathsf{t} \neq \mathsf{T} \wedge \mathsf{clock(t)} = \mathsf{c}$ **then** ▷ *Write–read conflict?*
3:     **if** deadlocked **then**
4:         **throw** consistency exception
5:     **else**
6:         Retry from line 1
7: $\mathsf{T.readLog} \leftarrow \mathsf{T.readLog} \cup \{\langle \mathsf{x}, \mathsf{v} \rangle\}$

Algorithm 11 BOUNDARY [Valor-A]: T's region ends

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

B. Correctness of Snappy

This section provides arguments that Snappy (in particular, Snappy-P) soundly and precisely provides SIx.

Theorem 1. ***Snappy is sound:*** *If an execution completes without deadlock under Snappy, the execution conforms to snapshot isolation of synchronization-free regions (SI).*

We have not proved this theorem; instead we provide an argument for its correctness. Following from prior work, FastRCD and FastRCD-A ensure that executions that complete without exception or deadlock provide *conflict serializability*, a sufficient condition for region serializability (Section 5.1) [14]. The difference between FastRCD-A and Snappy is that Snappy allows execution to proceed past a read–write conflict until the region end, at which point execution can proceed if the reader region has finished executing, even if it is waiting on read–write conflicts.

Consider an execution that completes without deadlock. Let R_i be an ongoing region with a read to x, followed by a write to x by region R_j. Snappy detects the read–write conflict and allows R_j to continue executing until region end, at which point R_j's thread waits until R_i reaches its region end. Thus, R_j cannot finish waiting at its region end until R_i ends.

According to the definition of conflict SI, $s_i \prec_t e_j$ due to the read–write conflict. Our soundness concern here is that the execution might also establish $e_j \prec_t s_i$ (which would violate conflict SI since $\prec_t$ is a partial order). However, if we suppose $e_j \prec_t s_i$, then there must exist a (potentially transitive) write–write or write–read conflict from R_j to R_i. Snappy in that case will ensure that R_i does not reach its end until R_j finishes waiting at its region end, which implies a deadlock due to the conclusion above about R_j waiting on R_i. A deadlock contradicts the original assumption that the execution completes without deadlock. We thus conclude that Snappy's relaxation for read–write conflicts does not lead to violations of SI.

Lemma 1. *If an execution deadlocks under Snappy, it has a write–write or write–read conflict.*[6]

Proof. Suppose an execution deadlocks under Snappy, but the execution has no write–write or write–read conflicts. Since Snappy waits at program writes and reads only if there is a write–write or write–read conflict (Algorithms 5 and 6), Snappy does not wait at program reads or writes for this execution.

The execution thus deadlocks by waiting at Snappy's only other waiting point, a region boundary (lines 3 in Algorithm 4). Let T be a thread that is waiting at a region boundary as part of a deadlock. According to the wait condition, $\mathsf{clock(t)} = \mathsf{c}$. The value c comes from a $\mathsf{t} \mapsto \mathsf{c}$ entry in $\mathsf{T.waitMap}$ (line 10 in Algorithm 5), which in turn comes from a $\mathsf{t} \mapsto \mathsf{c}$ entry from $\mathcal{R}_x$ (line 8 in Algorithm 6). $\mathcal{R}_x[\mathsf{t}]$ comes from $\mathsf{clock(t)}$, which must be odd because t is executing a region. So c must be odd in the wait condition at T's region boundary.

Since $\mathsf{clock(t)} = \mathsf{c}$, $\mathsf{clock(t)}$ is also odd, implying that thread t is in the middle of executing a region (i.e., not at a region boundary). Since Snappy waits only at region boundaries for this execution, t's region eventually finishes, incrementing $\mathsf{clock(t)}$ and thus negating thread T's waiting condition, i.e., $\neg(\mathsf{clock(t)} = \mathsf{c})$. This contradicts the earlier conclusion that T is waiting as part of a deadlock. □

Theorem 2. ***Snappy is precise:*** *If an execution deadlocks under Snappy, it has a region conflict and a data race.*

Proof. Suppose an execution deadlocks under Snappy. By Lemma 1, the execution has a region conflict, which is a sufficient condition for a data race. □

Acknowledgments

We thank Man Cao, Aritra Sengupta, Jake Roemer, Rui Zhang for helpful discussions.

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.

[6] Section 2.1 defines a region conflict.

[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 Consistency 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. 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.

[16] 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.

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

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

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

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

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

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

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

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

[25] M. Christiaens and K. D. Bosschere. Accordion Clocks: Logical Clocks for Data Race Detection. In *Euro-Par*, pages 494–503, 2001.

[26] M. Christiaens and K. De Bosschere. TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs. In *Symposium on Java Virtual Machine Research and Technology Symposium*, pages 15–15, 2001.

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

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

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

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

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

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

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

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

[35] 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.

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

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

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

[39] 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.

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

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

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

[43] 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.

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

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

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

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

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

[49] 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.

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

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

[52] 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.

[53] 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.

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

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

[56] 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.

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

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

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

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

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

[62] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, 1979.

[63] PCWorld. Nasdaq's facebook glitch came from race conditions, 2012. `http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

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

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

[66] 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*, page pages, 1997.

[67] 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.

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

[69] 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.

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

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

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

[73] 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.

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

[75] 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.

[76] 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.

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

[78] 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.

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

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

[81] 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.

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

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