Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability *

Aritra Sengupta Swarnendu Biswas Minjia Zhang Michael D. Bond Milind Kulkarni
Ohio State University Purdue University
{sengupta,biswass,zhanminj,mikebond}@cse.ohio-state.edu milind@purdue.edu

Abstract

Data races are common. They are difficult to detect, avoid, or eliminate, and programmers sometimes introduce them intentionally. However, shared-memory programs with data races have unexpected, erroneous behaviors. Intentional and unintentional data races lead to atomicity and sequential consistency (SC) violations, and they make it more difficult to understand, test, and verify software. Existing approaches for providing stronger guarantees for racy executions add high run-time overhead and/or rely on custom hardware.

This paper shows how to provide stronger semantics for racy programs while providing relatively good performance on commodity systems. A novel hybrid static–dynamic analysis called *EnfoRSer* provides end-to-end support for a memory model called *statically bounded region serializability* (SBRS) that is not only stronger than weak memory models but is strictly stronger than SC. EnfoRSer uses static compiler analysis to transform regions, and dynamic analysis to detect and resolve conflicts at run time. By demonstrating commodity support for a reasonably strong memory model with reasonable overheads, we show its potential as an always-on execution model.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Run-time environments

Keywords Dynamic analysis; static analysis; region serializability; memory models; atomicity; synchronization

1. Introduction

Shared-memory parallel programs have many possible behaviors due to the ways in which threads' memory accesses can interleave and can be reordered by compilers and hardware. Modern languages and systems provide a strong guarantee for any well-synchronized, or *data-race-free*, execution: its synchronization-free regions (SFRs) appear to execute atomically. However, executions with data races have few or no guarantees [1, 7, 8, 39], leading to unexpected, erroneous behaviors. Furthermore, handling the possibility of data races complicates static and dynamic analyses. Prior work has proposed stronger memory models, but they have lacked a compelling tradeoff between costs and benefits.

This paper's goal is to provide end-to-end support for a memory model that achieves a compelling balance between the benefits of stronger guarantees and the costs of providing those guarantees. Inspired by ideas from prior work on using *region serializability* (RS) to provide SC [4, 23, 37, 40], we focus on a new memory model called *statically bounded region serializability* (SBRS) that enforces serializability (atomicity) of statically bounded regions: regions bounded by synchronization operations, loop back edges, and method calls. SBRS offers an interesting cost–benefit tradeoff: it is strictly stronger than SC, yet we show that the bounded nature of its regions offers opportunities for static–dynamic analysis to enforce *end-to-end* SBRS with reasonable performance on commodity systems.

This paper presents a hybrid static–dynamic analysis for enforcing end-to-end SBRS called *EnfoRSer*. EnfoRSer enforces SBRS through (i) a static, intraprocedural compiler pass that partitions code into statically bounded regions, and transforms and instruments each region so that (ii) a runtime system can guarantee that regions appear to execute atomically. The key to EnfoRSer's operation is the interaction between the static transformations and the runtime system.

At a high level, EnfoRSer guarantees region atomicity using two-phase locking: each thread acquires lightweight, mostly-synchronization-free reader–writer locks before each memory access and does not release locks until the region ends. We present two distinct approaches for implementing this basic technique while overcoming its inherent tendency to deadlock. The first approach *executes regions idempotently* by deferring side effects (stores) until the region has acquired all of its locks. The second approach *executes regions speculatively*, rolling back speculative state when regions potentially conflict. While these approaches are related to *software transactional memory* (e.g., [25]), statically bounded regions enable a significantly different,

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

ASPLOS'15, March 14–18, 2015, Istanbul, Turkey.

ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694379

higher-performance design based on hybrid static–dynamic analysis (Section 9).

We have implemented EnfoRSer in a high-performance Java virtual machine and applied it to benchmarked version of large multithreaded applications. EnfoRSer adds 43 and 36% overhead on average for the idempotent and speculation approaches, respectively—without relying on custom hardware or whole-program analysis. After we extend EnfoRSer to use the results of whole-program static analysis that identifies definitely data-race-free (DRF) accesses [44], it adds 32 and 27% average overhead for the idempotent and speculation approaches, respectively. Thus, enforcing SBRS can incur reasonable overhead even in commodity systems, meaning that it is a feasible always-on memory model that could improve software reliability today—and perhaps be supported more efficiently by future hardware systems.

2. Background and Motivation

An execution is *data race free* (DRF) if every pair of conflicting accesses (i.e., accesses to the same variable where at least one is a write) is well synchronized [3]—meaning they are ordered by the *happens-before* relation, a partial order that is the union of program and synchronization order [28].

The DRF0 memory model. Modern shared-memory languages such as Java and C++ use variants of the *DRF0* memory model [2, 8, 39]. DRF0 provides a strong guarantee for any DRF execution: *serializability* (i.e., atomicity) of *synchronization-free regions* (SFRs).

However, DRF0 provides weak guarantees for racy executions. Java tries to preserve the memory and type safety of racy executions [39], although the resulting memory model has flaws [58]. C++ provides essentially no guarantees [8].

Eliminating data races effectively and efficiently is a challenging, unsolved problem (e.g., [11, 21, 43]). Moreover, developers often avoid synchronization in pursuit of performance, deliberately introducing data races that lead to unexpected, ill-defined behaviors. Adve and Boehm and Ceze et al. argue that languages and hardware must provide stronger memory models to avoid impossibly complex semantics [1, 7, 14]. 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." They "call upon software and hardware communities to develop languages and systems that *enforce* data-race-freedom . . . " [1].

Existing stronger memory models. Under the *sequential consistency* (SC) memory model, operations appear to interleave in an order respecting program order [29]. Providing end-to-end SC involves limiting memory access reordering by both the compiler and hardware, which slows programs and/or relies on custom hardware support [1, 32, 33, 41, 47, 51, 53, 55] (Section 9).

A promising approach for providing SC (and/or for detecting data races) is to provide *region serializability* (RS) of dynamically executed regions of code [4, 18, 23, 37, 38, 40, 52]. RS can be an advantageous mechanism since it can avoid unduly limiting compiler and hardware reordering of accesses. However, prior work that uses RS for SC relies on new hardware, adds high overhead, and/or *checks* SC instead of enforcing it, risking unexpected failures (Section 9).

While SC is stronger than DRF0-based memory models, it is not particularly strong. Under SC, it is still difficult and unnatural for programmers to reason about all interleavings, and for program analyses and runtime support to deal with all interleavings. Some operations that many programmers expect to execute atomically do not execute atomically under SC (e.g., 64-bit integer accesses in Java; multi-access operations such as $\mathsf{x}{+}{+}$ or $\mathsf{buffer}[\mathsf{index}{+}{+}] = \ldots$). Adve and Boehm argue that "programmers do not reason about correctness of parallel code in terms of interleavings of individual memory accesses, and sequential consistency does not prevent common sources of concurrency bugs . . . " [1].

While most work on RS-based memory models has focused on bounded regions, some work supports RS of regions bounded only by synchronization operations, a memory model we call *full-SFR RS* [37, 45]. While full-SFR RS is clearly a strong memory model, it may not be practical to enforce it—even with custom hardware support. Existing approaches rely on complex custom hardware and check (rather than enforce) full-SFR RS [37], or they rely on page-protection-based speculation and slow programs by two times or more (unless extra cores are available) [45].

3. Statically Bounded Region Serializability

This paper seeks to provide a strong memory model that can be supported with reasonable efficiency in commodity systems. Inspired by prior work that provides RS, our approach is based on enforcing RS. While providing full-SFR RS is appealing, it seems inherently difficult to enforce without heavyweight support for conflict detection and speculation, which would be expensive in software and complex in hardware. Thus, we seek a compromise memory model that balances strength and practicality.

Our choice is guided by two insights. First, enforcing RS of *dynamically bounded* regions (i.e., each executed region performs a bounded number of operations) permits simple, conservative region conflict detection, since the cost of misspeculation is bounded. Second, enforcing RS of not only dynamically but also *statically bounded* regions (i.e., an executing region executes each static instruction at most once) enables powerful static compiler transformations for enforcing region atomicity. Based on these insights, we propose to enforce a memory model called *statically bounded region serializability* (SBRS), which provides RS of statically bounded regions—regions demarcated at loop back edges and method calls as well as synchronization operations.

Section 8 evaluates SBRS's efficacy at avoiding real program errors. Here we discuss potential advantages and disadvantages of SBRS relative to other memory models.

Potential advantages and disadvantages. SBRS makes software automatically more reliable than under DRF0-based models. In addition to providing SC and thus avoiding

hard-to-reason-about SC violations (e.g., double-checked locking errors), SBRS eliminates all violations of atomicity of statically bounded regions, e.g., common patterns such as improperly synchronized read–modify–write accesses and accesses to multiple variables. The following code snippets—which programmers may already expect to execute atomically—will execute atomically under SBRS:

- `x += 42` (read–modify–write)
- `if (o != null) { ... = o.f; }` (check before use)
- `buffer[pos++] = value` (multi-variable operation)

Furthermore, because SBRS restricts possible behaviors, the job of various static and dynamic analyses or runtime systems can be simplified by assuming SBRS. Current analyses and runtimes either (unsoundly) ignore the effects of possible data races,[1] or they consider the effects but incur increased complexity or overhead. The model checker CHESS must consider many more possible executions because of the effects of data races [12, 42]. Static dataflow analysis must account for racy interleavings to be fully sound [16]. The primary performance challenge of software-based multithreaded record & replay is dealing with the possibility of data races [56]: RecPlay assumes data race freedom unsoundly to reduce costs [49]; Chimera shows that handling data races leads to prohibitively high overhead [30]; other approaches sidestep this problem but incur other disadvantages or limitations such as not supporting both online and offline replay [31, 46] or requiring extra available cores [56].

We note that code expansion optimizations, such as method inlining and loop unrolling, increase the size of statically bounded regions beyond those anticipated at the source-code level, eliminating more atomicity violations and increasing the scope of possible compiler reordering optimizations. Since code expansion makes regions strictly larger than they appear to be at the source-code level, the atomicity of source-code-level regions still holds.

SBRS is clearly not as strong as full-SFR RS, which will avoid more errors and provide fewer interleavings for static and dynamic analyses to consider. Programmers might find it easier to reason about full-SFR RS, which requires considering only synchronization operations, than SBRS, which requires thinking about method and loop boundaries as well. On the other hand, full-SFR RS requires interprocedural reasoning: a region is synchronization free only if (transitively) all of its callee methods are synchronization free.

In any case, we suspect that SBRS and full-SFR RS are mainly useful *not* for helping programmers reason about their code, but rather for enforcing behaviors that many programmers *already assume*.

Extending SBRS. While we expect most programmers can and should be unaware of runtime support for SBRS, experts might want more control. They could mark code regions as atomic, e.g., with `atomic {}` blocks. EnfoRSer could naturally enforce atomicity of marked, statically bounded regions, and use method inlining and loop unrolling on larger regions to allow EnfoRSer to execute them atomically. The largest regions could be handled by using software transactional memory (STM) integrated with EnfoRSer. These extensions are beyond this paper's scope.

Progress. If SC guarantees progress[2] for a program, SBRS guarantees progress for that program. A proof sketch of this claim is available in an extended technical report [50].

A program can be *unable* to make progress under SBRS even though it *might* make progress under SC. For example, the following program *might* terminate—but is *not guaranteed* to terminate—under SC:

Initially `x = false, y = false, done = false`

```
// T1:                    // T2:
while (!done) {           while (true) {
  x = !x;                   if (x != y) break;
  y = !y;                 }
}                         done = true;
```

In contrast, under SBRS, each loop body executes atomically, so the program *cannot* terminate.

Interestingly, *full-SFR RS* can impede the progress of a program for which SC guarantees progress, although full-SFR RS guarantees progress for any program for which *DRF0* guarantees progress. For example, the following program always terminates under SC; cannot terminate under full-SFR RS (any terminating execution would violate full-SFR RS); and may or may not terminate under DRF0 (e.g., the compiler can legally hoist each load out of its loop):

Initially `x = false, y = false`

```
// T1:          // T2:
y = true;       x = true;
while (!x);     while (!y);
```

Under SBRS, this program always terminates because regions are dynamically bounded.

4. Hybrid Static–Dynamic Analysis

This section describes EnfoRSer, a hybrid static–dynamic analysis that enforces end-to-end SBRS.

4.1 Overview

To enforce SBRS, EnfoRSer's compiler pass partitions the program into regions bounded by synchronization operations, method calls, and loop back edges. Dynamic analysis then enforces atomicity of regions. A naïve approach for enforcing atomicity is as follows. First, associate a "lock" with each potentially shared object[3] (e.g., by adding a header word to each object that tracks the object's ownership state). Henceforth, this paper uses the term "lock" to refer to per-objects locks that are added by EnfoRSer and acquired by its

[1] In theory, a static or dynamic analysis for C++ can provide any behavior for racy executions and still be sound, since C++ (unlike Java) provides no semantics for racy executions [8]. In practice, real-world programs have data races, so it is useful for analyses to behave sensibly in their presence.

[2] We assume a fair scheduler that does not allow any thread to starve.

[3] This paper uses the term "object" to refer to any unit of shared memory.

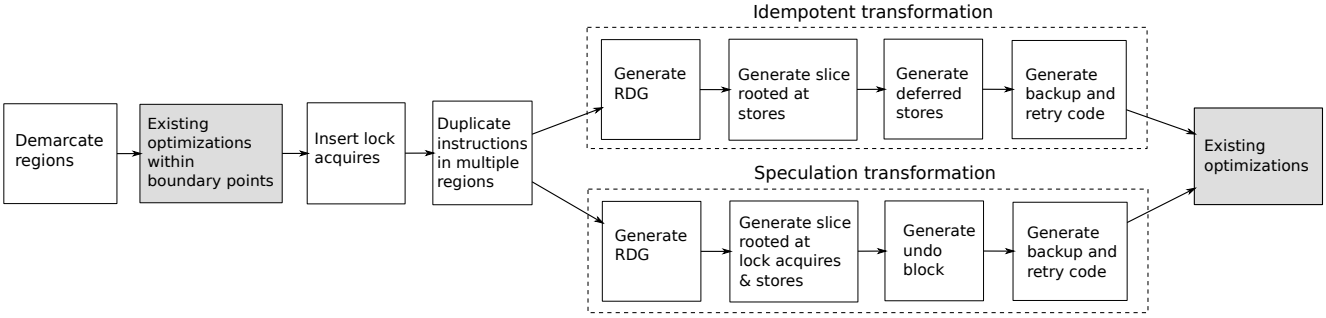

Figure 1. The transformations performed by an EnfoRSer-enabled optimizing compiler. Boxes shaded gray show existing compiler optimizations. Dashed boxes indicate transformations performed once on each region.

instrumentation and are *not* visible to programmers. Second, instrument the region to ensure that (1) a thread holds a lock on an object before the region accesses it and (2) the region releases locks only when it ends. This approach implements two-phase locking, so the execution will be equivalent to a serialized execution of regions.

This naïve approach suffers from two drawbacks. First, because threads can access objects in arbitrary orders, and per-object locks must be held until the end of the region to satisfy two-phase locking, a thread can deadlock while executing a region that attempts to acquire multiple locks. EnfoRSer solves this problem by applying a compile-time *atomicity transformation*: if a thread cannot acquire a lock because another thread holds it—we call this situation a "conflict," as all region conflicts result in a failed lock acquire—the transformed region is restarted, avoiding the deadlock. Second, acquiring a lock on each object access seems to require an atomic operation such as compare-and-swap (CAS), which would add unreasonable overhead. EnfoRSer uses special locks that mostly avoid atomic operations [10] (Section 4.3).

The key challenge is ensuring that a transformed region can be safely restarted without losing atomicity. If a region restarts after modifying shared objects, then either another thread could see those effects, or the region may behave incorrectly after restart. Either way, region atomicity is violated. We design, implement, and evaluate two atomicity transformations that address this challenge. (We investigate two different transformations in order to explore the space.)

The *idempotent* transformation modifies each region to defer stores until all per-object locks have been acquired. If a region encounters a conflict when acquiring a lock, it can safely restart since it executes idempotently up to that point. The challenge is generating correct code for deferring stores in the face of object aliasing and conditional statements.

The *speculation* transformation modifies the region to perform stores speculatively as it executes. The transformed region backs up the original value of stored-to memory locations. On a conflict, the modified region can restart safely by using the backed-up values to restore memory to its original state. The challenges are maintaining the backup values efficiently and restoring them correctly on retry.

Putting it all together, EnfoRSer's compiler pipeline operates as shown in Figure 1. First, it demarcates a program into regions (Section 4.2), and any optimizing passes in the compiler are performed, modified to operate within region boundaries. Next, the compiler inserts lock acquire operations before object accesses (Section 4.3). The compiler then performs a duplication transformation (Section 4.4) and builds an intermediate representation (Section 4.5), before transforming each region using one of the two atomicity transformations (Sections 4.6 and 4.7). Finally, any remaining compiler optimizations are performed.

At run time, if a thread executing a region detects a potential conflict, it restarts the region safely, preserving atomicity. The atomicity of regions guarantees equivalence to some serialized execution of regions.

4.2 Demarcating Regions

The first step in EnfoRSer's compiler pass is to divide the program into regions; later steps ensure that these regions execute atomically. The following program points are region *boundary points*: synchronization operations, method calls, and loop back edges.[4] Since method calls are boundary points, regions are demarcated at method entry and return, i.e., EnfoRSer's analysis is intraprocedural. An EnfoRSer region is a maximal set of instructions such that (1) for each instruction i, all (transitive) successors of i are in the region provided that the successors are reachable without traversing a region boundary point; and (2) there exists a "start" instruction s such that each instruction i in the region is reachable from s by only traversing edges in the region. Note that an instruction can be in multiple regions statically, i.e., reachable from multiple boundary points without an intervening boundary point—a situation that Section 4.4 addresses.

Reordering within and across regions. To ensure region atomicity, the compiler cannot be permitted to reorder instructions across region boundary points. Similar to DRFx's soft fences [40], EnfoRSer prohibits inter-region optimizations, by modifying optimization passes, such as common subexpression elimination, that may reorder memory accesses to ensure they do not reorder across boundary points. We find that prohibiting reordering across region boundaries

[4] A loop *back edge* is any control-flow edge to a loop header from a basic block dominated by the loop header. Loop *headers* are not boundary points.

impacts performance negligibly for our implementation; details are in an extended technical report [50].

4.3 Lightweight Reader–Writer Locks

The next step after demarcating regions is instrumenting program loads and stores with lock acquire operations. Acquiring traditional locks on every object access would be prohibitively expensive. EnfoRSer instead uses lightweight reader–writer locks that differ from traditional locks in two key ways. (1) A lock is always "acquired" in some state such as write-exclusive or read-shared, and acquiring the lock does not require an atomic operation if the thread already holds the lock in a compatible state. (2) A thread never automatically "releases" a lock; instead, another thread may acquire the lock, but it must first *coordinate* with the thread(s) that currently hold the lock. The design and implementation of these reader–writer locks (called "locks" for the remainder of the paper) are based closely on *Octet* locks [10]. We summarize their operation here.

Each object has a lock, denoted by o.lockState but not visible to programmers, that is always acquired in one of the following states:

- $\mathsf{WrEx_T}$: Thread T may read or write the object.
- $\mathsf{RdEx_T}$: T may read but not write the object.
- RdSh: Any thread may read but not write the object.

At each program store and load to the object referenced by o, the compiler inserts instrumentation denoted by acq_wr(o) and acq_rd(o), respectively, as the following pseudocode shows:

```
acq_wr(o); // instrumentation
o.f = ...; // program store

acq_rd(p); // instrumentation
... = p.g; // program load
```

The following pseudocode shows the definitions of acq_wr() and acq_rd() (T is the executing thread):

```
acq_wr(Object obj) {
  if (obj.lockState != WrEx_T)
    wrSlowPath(obj); /* change obj.lockState */
}

acq_rd(Object obj) {
  if (obj.lockState != WrEx_T &&
      obj.lockState != RdEx_T) {
    if (obj.lockState != RdSh)
      rdSlowPath(obj); /* change obj.lockState */
    load_fence; // see footnote 5
  }
}
```

To acquire a lock to obtain read or write access to its associated object, a thread T checks the state of the lock. If the lock's state is compatible with the access being performed (e.g., T wants to read or write an object in $\mathsf{WrEx_T}$ state), the lock is already "acquired" without any synchronization operations. This check is called the *fast path*.

Code path(s)	Transition type	Old state	Program access	New state	Sync. needed
Fast	Same state	$\mathsf{WrEx_T}$	R/W by T	Same	None
		$\mathsf{RdEx_T}$	R by T	Same	
		RdSh	R by T	Same	
Fast & slow	Upgrading	$\mathsf{RdEx_T}$	W by T	$\mathsf{WrEx_T}$	Atomic op.
		$\mathsf{RdEx_{T1}}$	R by T2	RdSh	
	Conflicting	$\mathsf{WrEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	Roundtrip coord.
		$\mathsf{WrEx_{T1}}$	R by T2	$\mathsf{RdEx_{T2}}$	
		$\mathsf{RdEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	
		RdSh	W by T	$\mathsf{WrEx_T}$	

Table 1. State transitions for lightweight locks.

Otherwise, a thread encounters a lock in an incompatible state, and it must change the lock's state. Table 1 shows all possible state transitions.[5] The first three rows (*Same state*) correspond to the fast path. The remaining rows show cases that must change a lock's state. Collectively, all operations that change a lock's state are called the *slow path*. An *upgrading* transition (e.g., from $\mathsf{RdEx_T}$ to RdSh) expands the set of accesses allowed by the lock; it requires an atomic operation to change the lock's state.

The slow path triggers a *conflicting* transition in cases where the program access conflicts with accesses allowed under the lock's current state. Because other thread(s) might be accessing the object without synchronization, merely changing the lock's state—even with an atomic operation—is insufficient. Instead, the thread executing the conflicting transition, called the *requesting thread*, must *coordinate* with every other thread, called a *responding thread*, that has access to the object, to ensure the responding thread(s) "see" the state change. For $\mathsf{WrEx_T}$ and $\mathsf{RdEx_T}$ states, T is the responding thread; for RdSh states, every other thread is a responding thread.

A responding thread participates in coordination only at a *safe point*: a point that does not interrupt instrumentation–access atomicity (i.e., atomicity of lock acquire instrumentation together with the program access it guards). Conveniently, managed language virtual machines already place safe points throughout the code, e.g., at every method entry and loop back edge. Furthermore, all blocking operations (e.g., *program* lock acquires and I/O) must act as safe points. If a responding thread is actively executing program code, the requesting and responding threads coordinate *explicitly*: the requesting thread sends a request, and the responding thread responds when it reaches a safe point. Otherwise, the responding thread is at a blocking safe point, so the threads coordinate *implicitly*: the requesting thread atomically modifies a flag that the responding thread will later see. Finally, in either case, the requesting thread finishes changing the lock's state and proceeds with the access.

Thus, while traditional locks add synchronization at every access, the locks used by EnfoRSer add no synchronization

[5] EnfoRSer does not need nor use Octet's RdSh counter [10]. Instead, EnfoRSer ensures that reads to RdSh locks happen after the prior write by issuing a load fence on the RdSh fast path.

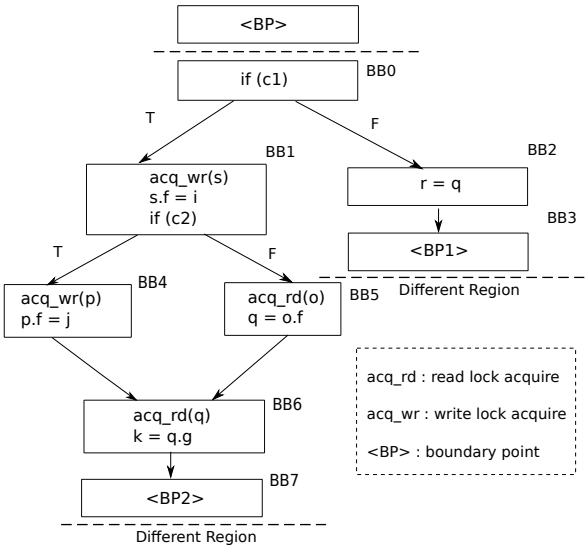

Figure 2. A region instrumented with lock acquire operations.

at non-conflicting accesses (the fast path), but they require coordination with other threads at conflicting accesses (the slow path). This tradeoff works well in practice for many programs, which aim for good thread locality by design.

Avoiding deadlock. As described so far, locks are deadlock prone: two threads each waiting to acquire a lock held by the other thread will wait indefinitely. To prevent deadlock, if a thread T is waiting for other thread(s) to relinquish a lock, T *allows other threads to acquire locks held by* T.

This behavior has interesting implications for EnfoRSer. When a thread tries to acquire an object's lock, it might give up *other objects' locks acquired in the same region*. As a result, rather than suffering from deadlock, as in traditional two-phase locking, regions may lose atomicity when they conflict. Hence, EnfoRSer's transformed regions actually restart *whenever a lock may have been released*—indicated by responding to another thread's coordination request.

Example. Figure 2 shows a region instrumented with lock acquire operations (acq_rd and acq_wr). The rest of this section uses this region as a running example.

4.4 Duplication Transformation

If a lock acquire (acq_rd(o) or acq_wr(o)) detects a potential conflict, it must retry the region by returning control to the region start. For the compiler to generate control flow for retry, the lock acquire must be statically in a single region.

The modified compiler thus performs a *duplication* transformation prior to atomicity transformations (Figure 1). This transformation duplicates instructions and control flow so that every instruction (not including boundary points) is in a single region. First, a simple dataflow analysis determines which boundary point(s) reach each instruction (i.e., which region(s) each instruction resides in). Second, in topological order starting at method entry, the duplication algorithm replicates each instruction that appears in $k > 1$ regions $k-1$ times, and retargets each of the original instruction's predecessors to the appropriate replicated instruction.

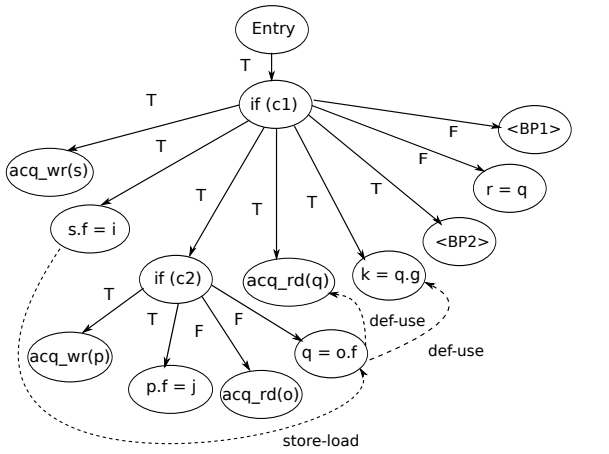

Figure 3. A region dependence graph for the region in Figure 2.

The duplication transformation does not duplicate any region boundary points. Since the number of region boundary points remains fixed, duplication cannot result in an exponential blowup in code size.

4.5 The Region Dependence Graph

In order to transform regions, the atomicity transformations require static *slices* [27]: the set of instructions that are data and control dependent on some instruction. Thus, for each region, an atomicity transformation first builds a *region dependence graph* (RDG), based on Ferrante et al.'s program dependence graph [20], which captures both data and control dependences in a region. The RDG needs to track true (write–read) dependences, but not output (write–write) or anti (read–write) dependences. It can ignore loop-carried dependences since regions are acyclic.

EnfoRSer's RDG construction algorithm uses a reaching-definitions analysis to compute intraprocedural data dependences in a region. The algorithm treats lock acquire operations acq_rd(o) and acq_wr(o) like a use of the object referenced by o. Because RDG construction is performed at the region level, aliasing relationships that arise due to operations outside the region cannot be inferred. Hence, the RDG construction algorithm adopts a conservative, type-based approach, e.g., it assumes that a load from field f of object o can have a dependence with any earlier store to the same field f of object p, since p and o could potentially alias. It also conservatively assumes that loads from arrays may have dependences from stores to type-compatible arrays. The construction algorithm uses the data dependences computed by the reaching-definitions analysis, as well as control dependences computed by a standard post-dominance analysis, to construct the final RDG.

Figure 3 shows the RDG for the region from Figure 2. The dotted lines depict store–load dependences (between accesses to object fields or array locations) and def–use dependences (between definitions and uses of local variables). The construction algorithm's aliasing assumptions lead to a store–load dependence between s.f = i and q = o.f, since s and o may be aliased. The Entry node is the root of the RDG

and has control dependence edges marked as T to all nodes (only one in this case) that are not control dependent on any other nodes. Other edges labeled T and F are control dependence edges from conditionals to dependent statements.

Slicing of the RDG. Each atomicity transformation must identify which parts of the region are relevant for the transformation. The transformation performs static program slicing of the RDG (based on static program slicing of the program dependence graph [27]) to identify relevant instructions. A *backward* slice *rooted* at an instruction i includes all instructions on which i is *transitively* dependent. Dependences include both data (store–load and def–use) and control dependences. A slice rooted at a set of instructions is simply the union of the slices of each instruction.

4.6 Enforcing Atomicity with Idempotence

The *idempotent* transformation defers side effects (program stores) until the end of the region. The side-effect-free part of the region, which includes all lock acquires, executes *idempotently*, so it can be restarted safely if a lock acquire detects a potential region conflict.

To defer program stores, the transformation replaces each static store with a definition of a new local variable. Figure 4 shows the region from Figure 2 after the idempotent transformation. The stores s.f = i and p.f = j from Figure 2 are replaced with assignments to fresh locals i' and j' in BB1 and BB3 of Figure 4. The two main challenges are (1) modifying loads that might alias with stores so that they read the correct value and (2) generating code at the end of the region that performs the stores that should actually execute.

Loads aliased with deferred stores. Any load that aliases a deferred store needs to read from the local variable that backs up the stored value, rather than from the deferred store's memory location. The RDG in Figure 3 includes a store–load edge from s.f = i to q = o.f because RDG construction conservatively assumes all type-compatible accesses might alias. As shown in BB1 of Figure 4, the transformation substitutes a definition of a new temporary variable i' = i in place of the store s.f = i. If s and o alias, then q = i' should execute in place of q = o.f; otherwise q = o.f should execute normally.

For each load, the transformation emits a series of alias checks to disambiguate the scenarios: one for each potentially aliased store that may reach the load. BB4–BB6 in Figure 4 show the result of this transformation. The transformation first emits the original load (BB4). Then, for each store on which the load appears to be dependent, the transformation generates an assignment from the corresponding local variable, guarded by an alias check (BB6, guarded by the check in BB5). By performing these checks in program order, the load will ultimately produce the correct value, even if multiple aliasing tests pass. If a possibly-aliased store executes conditionally, the transformed load is guarded by a conditional check in addition to the alias check to ensure that the store executed. A similar situation arises with array accesses, in which case the aliasing checks must not only compare the array references, but also the index expressions.

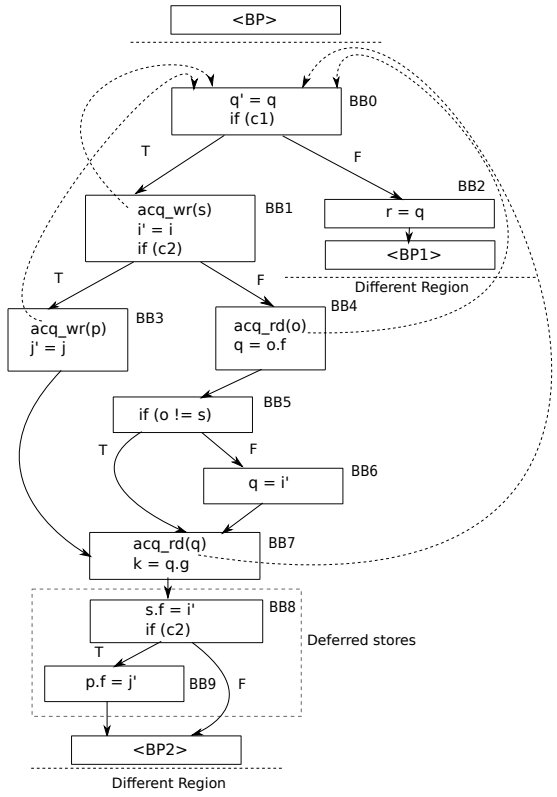

Figure 4. Region from Figure 2 after idempotent transformation.

Although this approach may generate substantial code per load, in practice later compiler passes can simplify it significantly. For example, if the compiler can prove two object references definitely do or do not alias, constant propagation and dead code elimination will remove unnecessary checks.

Deferred stores. It is nontrivial to generate code at the end of a region to perform the region's stores, because the code should perform only the stores that actually would have executed. The transformation generates code at each region exit based on a slice of the region's stores from the RDG. It generates each deferred store using the mapping from static stores to local variables. To ensure a deferred store only executes if it would have in the original region, the slice includes all condition variables on which stores are control dependent. The generated code checks these conditions before performing any deferred store. For example, BB8 in Figure 4 checks condition c2, executing the deferred store in BB9 only if the store would have occurred in the original region (note that the same condition guards j' = j in BB3).

Note that it is possible that the conditional variables guarding stores are overwritten during the execution of the idempotent portion of the region; if so, those conditionals may resolve differently while executing deferred stores. To avoid this problem, the transformation "backs up" the results of any conditional in the idempotent region in fresh local variables, and uses these locals in the conditionals guarding deferred stores. Similarly, any store to an object field or array in the original region is dependent on the base reference of

the object or array. If these base references may change during execution, the transformation backs them up at the point of the original store and uses the backups when executing the deferred stores.

Supporting retry. Although the idempotent transformation defers stores, the region can still have side effects if it modifies *local* variables that may be used in the same or a later region. For each region, the idempotent (and speculation) transformations identify such local variables and insert code to (1) back up their values in new local variables at region start and (2) restore their values on region retry. In Figure 4, local variable q is backed up in q' at the beginning of the region (BB0). If a lock acquire detects a potential conflict, the code restores the value of q from q' (not shown) and control returns to the region start (a dashed arrow indicates the edge is taken if the operation detects a potential conflict).

4.7 Enforcing Atomicity with Speculation

EnfoRSer's second atomicity transformation enables *speculative execution*. This transformation leaves the order of operations in the region unchanged, but it transforms the region to perform stores speculatively: before a store executes, instrumentation saves the old value of the field or array element being updated. If a lock acquire detects a possible conflict, the region restarts safely by "rolling back" the region's stores. Figure 5 shows the region from Figure 2 after applying the speculation transformation. The primary challenge is generating code that correctly backs up stored-to variables and (when a conflict is detected) rolls back speculative state.

Supporting speculative stores. Since regions are acyclic and intraprocedural, every static store executes at most once per region execution. Hence, the speculation approach's transformation can associate each store with a unique, new local variable. Before each store to a memory location (i.e., field or array element), the transformation inserts a load from the location into the store's associated local variable.

Generating code to roll back executed stores is complex due to the challenge of determining *which* stores have executed. The solution depends on the subpath executed through the region. To tackle this challenge, the transformation generates an *undo block* for each region. In reverse topological order, it generates an undo operation for each store s in the region that "undoes" the effects of s using the associated backup variable. To ensure that s is only undone if it executed in the original region, this undo operation executes conditionally. The transformation determines the appropriate conditions by traversing the RDG from s up through its control ancestors. As in the idempotent approach, conditional variables and object base references are backed up if necessary, so they can be used during rollback. The undo block in Figure 5 illustrates the result of this process. The store to p.f is topologically after the store to s.f, so it appears first in the undo block. The store to p.f is performed if both c1 and c2 are true, so the old value is restored under the same conditions. Likewise, the store to s.f is only performed, and hence only undone, if c1 is true.

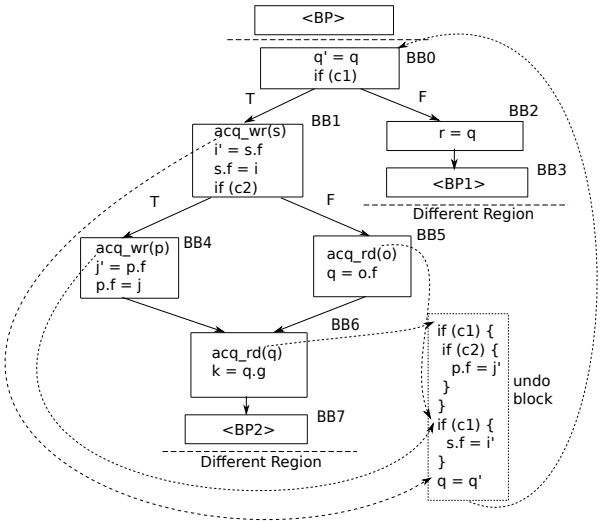

Figure 5. Region from Figure 2 after speculation transformation.

Supporting retry. The transformation generates control flow for each lock acquire to jump to the appropriate location in the undo block if the lock acquire detects a potential conflict. The jump target is the first undo operation associated with a store that is a control-flow predecessor of the lock acquire. Figure 5 shows these conditional jumps with dashed lines. As with the idempotent transformation, local variables must be backed up and restored during retry. In the figure, q is backed up in q' at the beginning of the region, and restored at the end of the undo block.

5. Optimizations

EnfoRSer's compiler analysis identifies accesses that do not need lock acquires, which in turn enables reducing the size of regions that need to be analyzed and transformed.

Statically redundant lock acquires. A lock acquire for an object access is "redundant" if each incoming path definitely performs a sufficiently strong lock acquire on the same object in the same region. An acq_rd() is redundant if definitely preceded by acq_rd() or acq_wr() on the same object; an acq_wr() is redundant if definitely preceded by acq_wr() on the same object. A lock is guaranteed to be in a compatible state at a redundant lock acquire; another thread cannot change the state without triggering restart of the current region. The compiler uses an intraprocedural, flow-sensitive dataflow analysis (based on an analysis used by prior work [10]) to identify redundant lock acquires.

Static data race detection. EnfoRSer's transformations can optionally use the results of sound static analysis that identifies all accesses that might be involved in data races. Remaining accesses are definitely data race free (DRF). An access that is DRF does not need a corresponding lock acquire. Prior work has also leveraged sound static race detection to simplify dynamic analysis [15, 19, 30, 57].

Whole-program static analysis is somewhat impractical in the real world since it relies on all code being available and all call graph edges being known at analysis time, which is

difficult in the presence of dynamic class loading and reflection. An implementation could handle unexpected dynamic behavior by dynamically recompiling *all* code to treat all accesses as potentially racy, or by using incremental static analysis to identify and recompile new potentially racy accesses. Our experiments sidestep this challenge by making all code and calls known to the static analysis.

Optimizing region demarcation. We optimize region demarcation to take advantage of optimizations that remove lock acquires. Optimized region demarcation distinguishes between *programmatic regions*—regions delimited at boundary points—and *enforced regions*—the regions whose atomicity is explicitly enforced by EnfoRSer's atomicity transformations. Optimized region demarcation starts and ends an enforced region at the first and last lock acquire of a programmatic region, respectively. Because any memory accesses outside of these boundaries are guaranteed to be DRF or guarded by locks earlier in the region, providing atomicity for these enforced regions automatically guarantees atomicity for programmatic regions.

An access that has no lock acquire but is inside of an enforced region must still be handled by the atomicity transformations: the idempotent transformation handles every possible store–load dependence in an enforced region, and the speculation transformation backs up every store in an enforced region. Although the endpoint of an enforced region may come before the end of a programmatic region, EnfoRSer does not treat the end of an enforced region as a safe point, so a thread will not release locks between the end of the enforced region and the end of the programmatic region.

6. Implementation

We have implemented EnfoRSer in Jikes RVM 3.1.3, a high-performance Java virtual machine [5] that provides performance competitive with commercial JVMs. We have made our implementation publicly available.[6]

Although the implementation targets a managed language VM, it should be possible to implement EnfoRSer's design for a native language such as C or C++. The main challenge would be adapting Octet to a native language [10].

Modifying the compilers. Jikes RVM uses two just-in-time dynamic compilers that perform method-based compilation. The first time a method executes, the *baseline* compiler compiles it directly from Java bytecode to machine code. If a method becomes hot, the *optimizing* compiler recompiles it at successively higher levels of optimization. The optimizing compiler performs standard intraprocedural optimizations. It performs aggressive method inlining but does not otherwise perform interprocedural optimizations.

We modify the optimizing compiler to demarcate regions and restrict reordering across regions throughout compilation; to insert lock acquires (we use the publicly available Octet implementation of lightweight locks [10]); and to perform EnfoRSer's atomicity transformations on the optimizing compiler's intermediate representation (IR).

[6] `http://www.jikesrvm.org/Research+Archive`

The baseline compiler does not use an IR, making it hard to implement atomicity transformations. Instead, in baseline-compiled code only, our implementation *simulates* the cost of enforcing SBRS without actually enforcing SBRS. The baseline compiler inserts (1) lock acquires, (2) instrumentation that logs each store in a memory-based undo log, and (3) instrumentation that resets the undo log pointer at each boundary point. This approach does not soundly enforce SBRS since it does not perform rollback on region conflicts. Since conflicts are infrequent and (by design) a small fraction of time is spent executing baseline-compiled code, this approach should closely approximate the performance of a fully sound implementation.

The compiler performs EnfoRSer's transformations on application and library code. Since Jikes RVM is written in Java, in a few cases VM code gets inlined into application and library code. Our prototype implementation cannot correctly handle inlined VM code that performs certain low-level operations or that does not have safe points in loops. To execute correctly, we identify and exclude 25 methods across all benchmarks and 10 methods in the Java libraries from EnfoRSer's transformations, instead inserting only lock acquires into these methods.

Demarcating regions. EnfoRSer demarcates regions at synchronization operations (lock acquire, release, and wait; thread fork and join), method calls, and loop back edges. These are essentially the same program points that are GC-safe points in Jikes RVM and other VMs. Since Jikes RVM makes each loop header a safe point, the implementation bounds regions at loop headers instead of back edges. To simplify the implementation, we currently bound regions along special control-flow edges for `switch` statements.

The optimizing compiler is able to identify all synchronization operations when it compiles application and library code, since synchronization in Java is part of the language. The implementation does not bound regions at `volatile` variable accesses, which does not affect progress guarantees.

Object allocations can trigger GC, so they are safe points and thus boundary points in our implementation. A production implementation could either define regions as being bounded at allocation (non-array allocations already perform a constructor call); defer GC past allocations; or speculatively hoist memory allocation above regions.

Retrying regions. A region must retry if another thread may have performed accesses that conflict with the region's accesses so far. This case can occur only if the region, while waiting for coordination, responds to coordination requests (Section 4.3). In the idempotent approach, lock acquires use this criterion to decide whether to retry a region. However, the speculation approach cannot easily use this criterion: a region must not lose access to objects until after it executes its undo block—at which point the region must re-execute—so the decision to retry the region must be made before the region responds to coordination requests and potentially loses needed access to an object. The speculation approach instead triggers retry whenever a lock acquire takes the slow

path. As a result, the idempotent approach provides lower retry rates than speculation—but the more precise retry criterion has a negligible performance impact (Section 7.3).

Retrying regions could lead to *livelock*, where conflicting regions repeatedly cause each other to retry. EnfoRSer could avoid livelock with standard techniques such as exponential backoff. Livelock is not an issue in our experiments, presumably because regions are short and conflicts are infrequent.

Runtime exceptions. A runtime exception (e.g., null pointer exception) can cause a region to exit early. This behavior can affect correctness for the idempotent approach since it reorders loads and stores, but we have not observed any problems in our experiments. We find that runtime exceptions are sufficiently rare (experimental details are in an extended technical report [50]) that just-in-time deoptimized recompilation could handle them efficiently.

7. Evaluation

This section evaluates EnfoRSer's run-time characteristics and performance.

7.1 Methodology

Benchmarks. The experiments execute our modified Jikes RVM on the multithreaded DaCapo benchmarks [6] versions 2006-10-MR2 and 9.12-bach (2009) with the large workload size (excluding programs Jikes RVM cannot run), distinguished by suffixes 6 and 9; and fixed-workload versions of SPECjbb2000 and SPECjbb2005.[7]

Platform. We build a high-performance configuration of Jikes RVM that adaptively optimizes the application as it runs (FastAdaptive) and uses the default high-performance garbage collector and adjusts the heap size automatically.

Experiments run on an AMD Opteron 6272 system with eight 8-core processors running Linux 2.6.32. We limit experiments to only four processors (32 cores) due to an anomalous result with 64 cores where EnfoRSer actually outperforms the baseline for some programs. (We find that adding *any* kind of instrumentation can improve performance, due to anomalies we have been able to attribute to Linux thread scheduling decisions [10].)

We have also evaluated EnfoRSer on an Intel Xeon E5-4620 system with four 8-core processors running Linux 2.6.32, in order to test EnfoRSer's sensitivity to the system architecture. On this platform, EnfoRSer adds nearly the same overhead as on the AMD platform (within 1% relative to baseline execution).

Static race detection. EnfoRSer can use any sound static analysis to identify definitely data-race-free (DRF) accesses (Section 5). We use the publicly available implementation of Naik et al.'s 2006 race detection algorithm, *Chord* [44]. By default, Chord is unsound (i.e., it misses races) because it uses a may-alias lockset analysis.[8] We thus *disable* lockset analysis, using only Chord's thread escape and thread fork–join analyses to identify DRF accesses.

The programs we evaluate use reflection and custom class loading, which present a challenge for static analysis. To handle reflection, we use a feature of Chord that executes the program to identify reflective call sites and targets. Chord does not run eclipse6 correctly in our environment, so EnfoRSer runs eclipse6 assuming all accesses are racy. jython9 has custom-loaded classes that present a challenge for Chord, so EnfoRSer assumes all accesses in custom-loaded classes are racy. We cross-checked Chord's results with a dynamic race detector's output; we found one class (in jbb2005) that Chord does not analyze at all (for unknown reasons), so EnfoRSer fully instruments this class.

Static analysis is a one-time cost, incurred only when the program changes. In any case, Chord's cost is low: analyzing any of the programs we evaluate takes at most 90 seconds.

[7] `http://www.spec.org/jbb200{0,5}`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

[8] We have confirmed that there is no available implementation of their 2007 algorithm, which uses conditional must-not-alias analysis to be sound [43].

	Threads		Insts.	Insts. /	Dyn. regions retried	
	Total	Live	executed	region	Idem.	Spec.
eclipse6	18	12	5.6×10^{10}	27	<0.01%	<0.01%
hsqldb6	402	102	5.7×10^{9}	32	<0.01%	0.50%
lusearch6	65	65	5.7×10^{9}	31	<0.01%	<0.01%
xalan6	9	9	8.5×10^{10}	29	0.05%	1.07%
avrora9	27	27	3.8×10^{10}	36	<0.01%	0.81%
jython9	3	3	8.5×10^{10}	29	<0.01%	<0.01%
luindex9	2	2	4.2×10^{9}	23	<0.01%	<0.01%
lusearch9	c	c	2.4×10^{9}	26	<0.01%	<0.01%
pmd9	5	5	1.2×10^{9}	22	<0.01%	0.11%
sunflow9	$2\times c$	c	3.0×10^{9}	26	<0.01%	0.03%
xalan9	c	c	9.9×10^{9}	23	0.37%	8.16%
pjbb2000	37	9	3.0×10^{9}	21	<0.01%	1.29%
pjbb2005	9	9	9.7×10^{9}	22	1.30%	17.91%

Table 2. Dynamic execution characteristics. A few programs launch threads proportional to the number of cores c, which is 32 in our experiments.

	Without static race detection					With static race detection				
	0	1	2–3	4–7	≥8	0	1	2–3	4–7	≥8
eclipse6	9%	25%	10%	38%	17%	9%	25%	10%	38%	17%
hsqldb6	7%	14%	11%	60%	8%	9%	13%	11%	59%	8%
lusearch6	5%	44%	14%	29%	8%	22%	43%	14%	19%	2%
xalan6	15%	27%	5%	30%	23%	45%	21%	9%	11%	13%
avrora9	6%	32%	5%	19%	38%	8%	31%	5%	18%	38%
jython9	24%	35%	5%	28%	8%	73%	18%	0%	8%	1%
luindex9	2%	28%	9%	35%	26%	23%	32%	8%	26%	11%
lusearch9	3%	42%	10%	32%	12%	24%	44%	10%	17%	5%
pmd9	5%	45%	7%	31%	12%	37%	30%	4%	20%	7%
sunflow9	18%	22%	16%	25%	19%	34%	30%	14%	13%	8%
xalan9	15%	33%	6%	21%	25%	42%	30%	4%	11%	12%
pjbb2000	42%	3%	26%	24%	4%	44%	23%	22%	7%	4%
pjbb2005	14%	17%	24%	37%	8%	18%	23%	14%	38%	7%

Table 3. Percentage of dynamic regions executed with various complexity (static accesses), without and with identifying statically DRF accesses to optimize region demarcation.

7.2 Run-Time Characteristics

The *Threads* columns of Table 2 report the total number of threads created and the maximum number of live threads.

The table's remaining columns report characteristics of executed regions, averaged across 10 trials. The *Insts.* columns report total dynamic IA-32 instructions and dynamic instructions per executed region, measured without

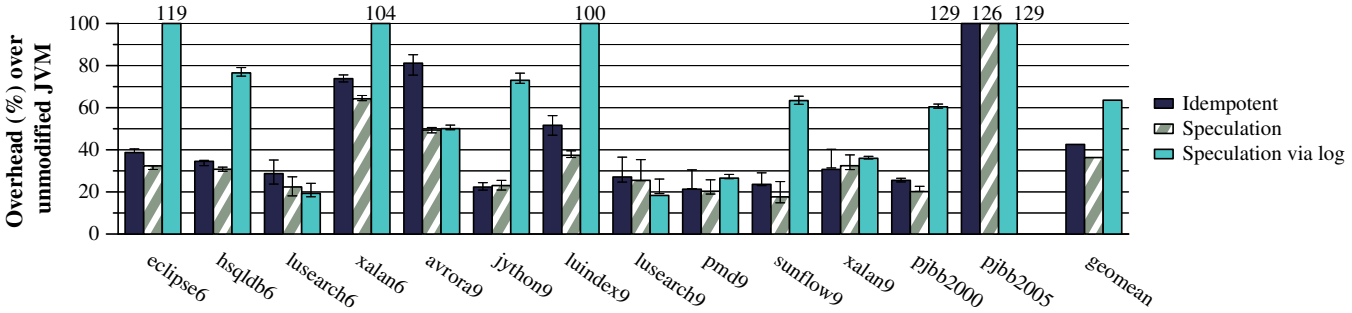

Figure 6. Run-time overhead over an unmodified JVM of providing SBRS with EnfoRSer's two atomicity transformations and speculation that uses a log to simulate a stripped-down version of STM.

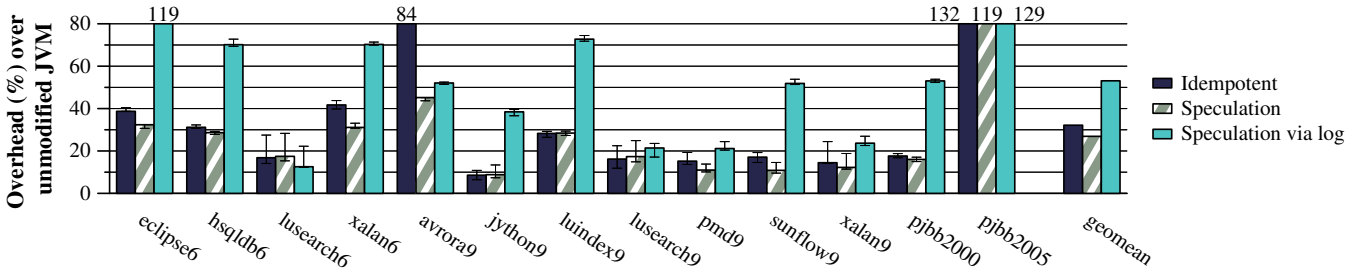

Figure 7. Run-time overhead over an unmodified JVM when transformations use whole-program static analysis to identify definitely data-race-free accesses. Otherwise, configurations are the same as Figure 6.

any EnfoRSer instrumentation. The programs each execute billions of instructions, divided into regions that execute 21–36 IA-32 instructions each on average. Across all programs, each region executes 27 instructions on average. EnfoRSer's regions are comparable in size to DRFx's statically bounded regions (about 10 instructions per region [40]).

The last two columns show that the vast majority of regions do not detect a conflict. The idempotent approach provides a substantially lower retry rate than the speculation approach, since the idempotent transformation uses a more precise retry check (Section 6), but we have found that making the check more precise does not significantly improve the idempotent approach's performance. The program pjbb2005 has the highest retry rate; unsurprisingly it has the highest rate of conflicts [10]. The high cost of coordination to handle these conflicts leads to high overhead for both the idempotent and speculation approaches (Section 7.3).

Table 3 evaluates the complexity of executed regions. We measure a region's complexity using the *static* count of accesses in the region, after performing the shrinking optimization from Section 5. Unsurprisingly, static data race detection helps simplify regions, i.e., executed regions tend to be less complex overall. Even so, all programs still rely on EnfoRSer's transformations for a significant number of complex regions (i.e., regions with $\geq$2 static accesses).

7.3 Performance

Figure 6 shows the overhead EnfoRSer's transformations add over unmodified Jikes RVM, *without* making use of static race detection. Each bar is the median of 15 trials (to minimize effects of machine noise); each bar shows 95% confidence intervals centered at the mean. The compiler duplicates instructions that are in multiple regions statically, adding 7% alone (not shown). The idempotent and speculation approaches add 43 and 36% total overhead, respectively.

The idempotent approach incurs costs to buffer and replay stores correctly and to check for aliasing at loads. Speculation is the better-performing approach since (in the common, non-conflicting case) it incurs only the cost of backing up each store's old value to a local variable. In a more aggressive compiler, the idempotent approach might have an advantage by enabling more intra-region reordering.

EnfoRSer adds the highest overhead for pjbb2005 (126–129%). This overhead primarily comes from the lightweight locks, which alone add 114% to pjbb2005 due to a relatively high conflicting access rate [10], which leads to expensive coordination among threads. EnfoRSer could achieve lower overhead for high-conflict executions by making use of "hybrid" lightweight locks that adaptively avoid coordination for high-conflict objects [13].

Static race detection. Figure 7 shows the same configurations when the transformations make use of statically identified DRF accesses. Identifying DRF accesses eliminates about a quarter of the overhead on average: the idempotent and speculation transformations enforce SBRS at 32 and 27% overhead, respectively. Although using static race detection seems to have a small adverse effect on idempotent performance for pjbb2005, this effect is likely due to high run-to-run variation for this program (we have confirmed that the confidence intervals overlap).

Log-based speculation. The *Speculation via log* configuration in Figures 6 and 7 measures a stripped-down version of STM that uses memory-based undo logs instead of local variables for backing up stores, but still relies on EnfoRSer's lightweight locks and efficient conflict detection. On average it adds 64 and 53% total overhead, without and with sound static race detection, respectively, significantly more than EnfoRSer's speculation approach. The memory-based log incurs these high costs despite adding just one of several costs that STMs typically incur over EnfoRSer (Section 9).

Compilation time. The overheads in Figures 6 and 7 naturally include the costs of just-in-time compilation. EnfoRSer slows the optimizing compiler by performing additional analyses and transformations, and by bloating the internal representation (IR) and slowing downstream passes. Since EnfoRSer's analysis is intraprocedural, its complexity scales well with program size. We find that compile time increases over unmodified Jikes RVM by 2.5X and 2.1X without static race detection, and 1.9X and 1.7X with static race detection, for the idempotent and speculation approaches, respectively. However, the effect of EnfoRSer's compilation overhead on overall execution time is modest: a few percent, relative to baseline execution time.

Scalability. EnfoRSer's overhead scales well when varying the number of application threads. Results are available in an extended technical report [50].

Summary. Overall, these results show that EnfoRSer enforces SBRS at a reasonable cost of 27% average overhead (using the speculation approach with sound static race detection). To our knowledge, this represents the lowest reported overhead of any kind of end-to-end RS enforcement on commodity systems. The best-performing prior work (which, admittedly, provides full-SFR RS) requires additional available cores to avoid adding over 100% overhead [45]—a fundamentally different, replication-based approach that would *not* benefit from identifying statically DRF accesses.

8. Avoiding Erroneous Behavior

This section describes a limited study of one of SBRS's potential benefits: its ability to automatically avoid errors caused by data races. To help expose such errors, we have implemented *adversarial memory* (AM), a dynamic analysis that helps expose behaviors that are allowed under the Java memory model [22, 39]. (Similar behaviors are possible under other language memory models including C++'s [8].) Under AM, each load from memory can choose from a buffer of stored values; the load may choose any value that does not violate established happens-before relationships. AM instruments potentially racy memory locations, identified by a sound dynamic race detector based on the FastTrack algorithm [9, 21].

We first identify potential errors by executing programs with AM. We use the DaCapo and SPECjbb benchmarks, as well as the smaller Java Grande benchmarks [54] evaluated by the AM paper [22]. Since the EnfoRSer implementation enforces SBRS only in Jikes RVM's optimizing compiler, our experiments forcibly compile all methods with the optimizing compiler. Table 4 reports a row for each error exposed by AM. AM exposes two different errors in SPECjbb2000. Each result is repeatable across many trials. The *JMM* column shows erroneous behaviors allowed under the Java memory model, exposed using AM.

	Erroneous behavior		
	JMM (AM alone)	**SC** (perturbation & inspection)	**SBRS** (AM + EnfoRSer)
hsqldb6	Infinite loop	None	None
sunflow9	Null ptr exception	None	None
jbb2000	Corrupt output	Corrupt output	None
jbb2000	Infinite loop	None	None
sor	Infinite loop	None	None
lufact	Infinite loop	None	None
moldyn	Infinite loop	None	None
raytracer	Fails validation	Fails validation	None

Table 4. Errors exposed by adversarial memory (AM) that are possible under the Java memory model, and whether the errors are possible under SC and SBRS.

To evaluate EnfoRSer's ability to avoid these errors, we use AM and EnfoRSer in the same execution. The compiler performs either of EnfoRSer's atomicity transformations, followed by AM's instrumentation of racy loads and stores. We integrate EnfoRSer and AM by making AM aware of the happens-before edges established by EnfoRSer's locks (e.g., coordination between threads). The *SBRS* column shows that EnfoRSer successfully avoids all erroneous behavior, by providing atomicity of statically bounded regions even in the presence of data races.

We have also determined whether these erroneous behaviors are possible under sequential consistency (SC). Despite the fact that these errors do not occur in typical runs (i.e., runs without AM), two of the errors are still possible under SC (the *SC* column):

SPECjbb2000 uses unsynchronized updates to a shared (64-bit) long field to keep track of elapsed time in milliseconds. Java makes no guarantees for the atomicity of accesses to the low and high 32 bits of a long [22, 34]. We confirmed an atomicity violation by inspecting the code. We were not able to reproduce a violation since it would take a very long time to overflow the counter's low bits (AM in fact exposes a *visibility* error by reading a stale value). SBRS avoids the errors since it inherently makes accesses to long fields atomic.

In raytracer, multiple threads perform additions to a shared int field called checksum1:

```
checksum1 = checksum1 + val; // val is thread local
```

Under SC, the program still computes an incorrect checksum if two threads' load–add–store operations interleave. SBRS eliminates the error by making the load–add–store atomic.

SC eliminates the other errors exposed by AM. We find that five out of six of these errors are *visibility* errors: a thread sees a "stale" value not possible under an SC execution. For each of these errors, a loop cannot terminate because

it repeatedly sees a shared variable's stale value instead of an up-to-date value. The sixth error is in sunflow9, which throws an exception when it reads and then dereferences a stale value of null. We have been unable to expose this error under SC (or SBRS), suggesting that it is an SC violation.

In general, many real-world atomicity violations are possible under SC (e.g., [35, 36, 61]). If the region that requires atomicity is small enough, SBRS eliminates the error.

9. Related Work

Section 2 covered related work on enforcing RS-based memory models. This section covers other approaches.

Software transactional memory. *Transactional memory* (TM) guarantees atomicity of programmer-annotated code regions [25, 26]. EnfoRSer's approach is analogous to using *software* TM (STM) [25] to provide atomicity of every statically bounded region. In particular, EnfoRSer's idempotent transformation behaves similarly to a lazy-versioning STM, while its speculation transformation behaves similarly to an eager-versioning STM. However, EnfoRSer provides atomicity much more efficiently than STM for three reasons:

1. STMs maintain read/write sets (i.e., the last transaction(s) to read and write each object) to detect precisely whether an access conflicts with another thread's ongoing transaction. In contrast, EnfoRSer avoids maintaining read/write sets, at the cost of false region conflicts. The rollbacks these false conflicts trigger incur only a small penalty since statically bounded regions are short.
2. STMs that use eager or lazy versioning maintain *undo logs* or *redo logs*, respectively. At each store, the STM appends an entry to the memory-based log containing the address of the stored-to location and its old or new value. EnfoRSer can instead map each store to a dedicated local variable for deferring or backing up the store, because regions are statically bounded.
3. The locks used by EnfoRSer are a lighter-weight mechanism for conflict detection than STMs typically use, as long as relatively few accesses conflict. (Recent work introduces an STM that uses similar lightweight locks [60].)

Hardware TM. Hammond et al. introduce a memory model and associated hardware where all code is in transactions [24]. However, manufacturers have been reluctant to incorporate general-purpose *hardware* TM (HTM) support into already-complex cache and memory subsystems.

Intel's recently released Haswell architecture provides *restricted transactional memory* (RTM) support with an upper bound on shared-memory accesses in a transaction [59]. It might seem at first that RTM would be well suited to enforcing SBRS. However, two studies find that the overhead of an RTM transaction is substantial: the startup and tear-down costs of each transaction are about the same as three compare-and-swap operations [48, 59]. In contrast, EnfoRSer's approach avoids atomic operations at most accesses, most likely achieving substantially lower overhead than an approach based on atomic operations or RTM transactions (except perhaps for high-conflict workloads), although an empirical comparison is beyond the scope of this paper. Future limited HTM implementations might provide lower-overhead transactions, enabling efficient support for SBRS. In the meantime, EnfoRSer can provide reasonably efficient support for SBRS in commodity systems.

Checking for region conflicts. Prior work *checks* region conflict freedom (RCF) in order to detect SC violations and/or data races [18, 23, 37, 40, 52]. Regions can be delimited only by synchronization operations [18, 37] or statically bounded to simplify the hardware [40, 52]. These approaches rely on custom hardware [23, 37, 40, 52] or add high overhead [18]. Furthermore, checking RCF can generate errors unexpectedly, even for executions for which RS is not violated. In contrast, EnfoRSer *enforces* RCF and thus RS in commodity systems.

Enforcing atomicity in hardware. Prior work relies on custom hardware support to enforce atomicity of regions. *AtomAid* avoids some atomicity violations [38], while *BulkCompiler* focuses on providing SC [4]. Their regions do not correspond to well-defined regions at the source-code level (which could help programmers and analyses), although they could potentially be made to do so.

Sequential consistency. Providing end-to-end SC requires restricting reordering in both the compiler and hardware. Whole-program static analysis called *delay sets* detects possible SC violations and introduces memory fences conservatively (e.g., [51, 55]).

Hardware can reorder loads and stores speculatively (e.g., [32, 33, 47]). Compilers can be modified to enforce SC, with additional hardware support providing end-to-end SC [41, 53]. Providing end-to-end SC—a weaker property than SBRS—seems destined to add nontrivial overhead due to restricting reordering by the compiler and hardware.

Idempotent execution. De Kruijf and Sankaralingam transform regions to execute idempotently [17]. Their approach targets hardware fault recovery and thus does not detect shared-memory conflicts. The approach handles def–use dependences but not store–load dependences—the primary challenge for EnfoRSer's idempotent transformation.

10. Conclusion

EnfoRSer leverages hybrid static–dynamic analysis to ensure that statically bounded, synchronization-free regions execute atomically on commodity systems. By providing reasonable efficiency in commodity systems, EnfoRSer can help make SBRS the default memory model, spurring development of new hardware for even faster SBRS support.

Acknowledgments

We thank our shepherd, Serdar Tasiran, and the anonymous reviewers for detailed and insightful suggestions for improving this work. Thanks to Hans Boehm, Man Cao, Meisam Fathi Salmi, Jipeng Huang, Brandon Lucia, Todd Millstein, Madan Musuvathi, Andrew Myers, Satish Narayanasamy, and Chris Stone for valuable discussions, suggestions, and other feedback; and to Mayur Naik for help with Chord.

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

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

[12] S. Burckhardt and M. Musuvathi. Effective Program Verification for Relaxed Memory Models. In *CAV*, pages 107–120, 2008.

[13] M. Cao, M. Zhang, and M. D. Bond. Drinking from Both Glasses: Adaptively Combining Pessimistic and Optimistic Synchronization for Efficient Parallel Runtime Support. In *WoDet*, 2014.

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

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

[16] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow Analysis for Concurrent Programs using Datarace Detection. In *PLDI*, pages 316–326, 2008.

[17] M. de Kruijf and K. Sankaralingam. Idempotent Code Generation: Implementation, Analysis, and Evaluation. In *CGO*, pages 1–12, 2013.

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

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

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *TOPLAS*, 9(3):319–349, 1987.

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

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

[23] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *SPAA*, pages 316–326, 1991.

[24] 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.

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

[26] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[27] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *PLDI*, pages 35–46, 1988.

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

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

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

[31] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*, pages 77–90, 2010.

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

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

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

[35] 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.

[36] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ASPLOS*, pages 37–48, 2006.

[37] 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.

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

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

[40] 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.

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

[42] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.

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

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

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

[46] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, pages 177–192, 2009.

[47] 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.

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

[49] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *TOCS*, 17:133–152, 1999.

[50] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. Technical Report OSU-CISRC-11/12-TR18, Computer Science & Engineering, Ohio State University, 2015. `http://www.cse.ohio-state.edu/~mikebond/papers.html#EnfoRSer`.

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

[52] 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.

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

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

[55] 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.

[56] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, pages 15–26, 2011.

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

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

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

[60] M. Zhang, J. Huang, M. Cao, and M. D. Bond. LarkTM: Efficient, Strongly Atomic Software Transactional Memory. In *PPoPP*, 2015. To appear.

[61] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam. ConAir: Featherweight Concurrency Bug Recovery via Single-threaded Idempotent Execution. In *ASPLOS*, pages 113–126, 2013.