Hybrid Static–Dynamic Analysis for 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

Ohio State CSE technical report #OSU-CISRC-11/12-TR18, updated November 2013

Abstract

Shared-memory parallel programs have many possible behaviors due to the many ways in which conflicting memory accesses can interleave. This plethora of behaviors complicates programmers' understanding of software, makes it more difficult to test and verify software, and leads to atomicity and sequential consistency (SC) violations. As prior work argues, providing *region serializability* (RS) makes software more reliable and simplifies programming and debugging. However, existing approaches that provide RS have serious limitations such as relying on custom hardware support or adding high overhead.

This paper introduces an approach called EnfoRSer that enforces RS of statically bounded regions. EnfoRSer is a hybrid static–dynamic analysis that performs static compiler analysis to transform regions, and dynamic analysis to detect and resolve conflicts at run time. We design, implement, and evaluate three distinct approaches for making regions execute atomically: one enforces mutual exclusion of regions, another executes idempotent regions, and the third executes regions speculatively. EnfoRSer provides RS of statically bounded regions completely in software without precluding compiler optimizations within regions. By demonstrating commodity support for RS with reasonable overheads, we demonstrate its potential as an always-on execution model.

1. Introduction

Parallel shared-memory programs have many possible behaviors due to the ways in which threads' memory operations can interleave or be reordered by compilers and hardware. Current languages and systems make strong semantic guarantees for well-synchronized, or *data-race-free* (DRF), executions. However, an execution with a data race can lead to unexpected, erroneous behaviors and complicate static and dynamic analyses that check or enforce concurrency correctness. Researchers have proposed stronger memory models such as sequential consistency (SC) [25], which constrains compiler and hardware reordering, yet still leaves many possible interleavings for programmers and analyses to reason about.

A stronger memory model that constrains interleavings and matches programmer expectations is *region serializability* (RS), in which execution appears equivalent to some serialization of synchronization-free regions [38]. Recent work by Ouyang et al. shows how to provide RS of synchronization-free regions in commodity systems by slowing programs by more than 2X [38].

This paper focuses on providing *RS of statically bounded regions*, a memory model in which regions are bounded by synchronization, method calls, and loop back edges. We argue that RS of statically bounded regions provides many of the same benefits as RS of full synchronization-free regions, while demonstrating that bounded regions offer opportunities for efficient runtime support. Programmers already expect statically bounded regions to execute atomically. RS of statically bounded regions prevents errors, such as some atomicity violations and all sequential consistency (SC) violations. Testing, model checking, and verification tools can check fewer paths if they can assume regions serialize (e.g., [35]). Runtime support such as multithreaded record & replay, deterministic execution, and dynamic bug detectors can be both sound and fast under RS (e.g., [27, 37]).

EnfoRSer: Efficient RS of Statically Bounded Regions

This paper presents EnfoRSer, an approach for enforcing RS of statically bounded regions in commodity systems. The key idea of EnfoRSer is to statically divide a program into regions and then, at run time, perturb execution so that regions execute atomically. EnfoRSer enforces RS through a hybrid static–dynamic analysis: (i) a static analysis and compiler pass that partition programs into regions and transform and instrument each region so that (ii) a runtime system can guarantee that regions execute atomically. The key to EnfoRSer's operation is the interaction between the static transformations and the runtime system's dynamic execution.

At a high level, the static compiler analysis divides the program into statically bounded, synchronization-free regions. It guarantees region atomicity using two-phase locking: acquiring lightweight, mostly-synchronization-free read/write locks before each memory access and not releasing locks until the region ends. We present three distinct approaches for implementing this basic technique while overcoming its inherent tendency to deadlock. The first approach *enforces mutual exclusion* of regions by hoisting all lock acquires to the start of the region, a transformation that is feasible because regions are statically bounded. The second approach *executes regions idempotently* by deferring side effects (stores) until the region has acquired all of its locks. The third approach *executes regions speculatively*, rolling back speculative state in the event of a possible region conflict. While the first approach is related to lock inference [13], and the second and third approaches are related to transactional memory [21, 22], statically bounded regions enable a significantly different, higher-performance design based on hybrid static–dynamic analysis.

We implement EnfoRSer's hybrid static–dynamic analysis in a high-performance Java virtual machine, and apply it to large multithreaded benchmarks. EnfoRSer adds 59, 47, and 40% overhead on average for the mutual exclusion, idempotent, and speculation approaches, respectively—without relying on custom hardware or whole-program analysis. This result compares favorably to the alternate approach for providing RS on commodity hardware, which slows programs by more than 100% (unless extra cores are available for speculative execution) [38]. These results show that RS of statically bounded regions can have 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. Motivation

A shared-memory execution is *data race free* (DRF) if all conflicting access pairs (i.e., accesses to the same variable where at least one is a write) are well synchronized—meaning they are ordered by the *happens-before* relation, a partial order that is the union of program and synchronization order [24].

The DRF0 memory model. Modern languages use the *DRF0* memory model or a variant of it [2, 8, 32]. The DRF0 model pro-

vides strong guarantees for a DRF execution: it provides serializability of synchronization-free regions, i.e., synchronization-free regions execute atomically.

However, the DRF0 memory model and its variants provide weak guarantees for racy executions. C++ provides essentially no guarantees, while Java preserves memory and type safety but otherwise allows interleaving and reordering in conflicting synchronization-free regions [8, 32]. Eliminating data races effectively and efficiently is a challenging, unsolved problem (e.g., [10, 17, 36]). Moreover, developers often avoid synchronization in pursuit of performance, deliberately introducing races that complicate reasoning about behavior. Recently, Adve, Boehm, and Ceze et al. have argued that languages and hardware must provide stronger memory models to avoid impossibly complex semantics [1, 7, 12].

Sequential consistency. An appealing alternative to DRF0 is one that provides stronger semantics for racy executions. *Sequential consistency* (SC) is a memory model in which operations appear to interleave in an order respecting program order [25]. SC is not a particularly strong model. It is still difficult and unnatural for programmers to reason about all interleavings, and for static and dynamic analyses to deal with all interleavings. And still, providing end-to-end SC slows programs nontrivially (e.g., [28, 34, 41, 44])—making it doubtful whether the tradeoff is worthwhile.

Region serializability. A stronger model is *region serializability* (RS), in which code regions appear to execute atomically regardless of whether there are data races. Recent work by Ouyang et al. provides RS of synchronization-free regions—regions bounded only by synchronization (as well as system calls since the approach is based on speculation) [38]. However, the approach slows programs by more than two times on average (without additional available cores for speculative execution).

This paper proposes a memory model that provides RS of *statically bounded regions*. Statically bounded regions are demarcated at loop and method boundaries as well as synchronization, providing bounded execution and also enabling powerful static compiler transformations. We argue that RS of statically bounded regions provides many of the same benefits as RS of full synchronization-free regions, and we show that it may be more feasible to enforce, particularly in commodity systems.

Programmers already assume RS at least to some extent, even though DRF0 does not guarantee this property. Our proposed memory model, RS of statically bounded regions, provides atomicity of regions demarcated at well-defined points: synchronization operations, calls, and loop back edges. These points are identifiable at the source code level, so an IDE can show programmers which regions will execute atomically during development, helping programmers see, for example, that adding a method call will break an atomic region into two regions. Although it may seem counterintuitive for programmers to have to reason about atomicity bounded at various program points, we note that RS of *full synchronization-free regions* presents a similar challenge: a method call splits a region only if the call tree performs synchronization, which is difficult to know at compile time, especially with virtual method dispatch and dynamic class loading. Moreover, it is not clear that providing larger regions aids programmability; regions that span multiple methods require programmers to reason about complex interprocedural behavior, so most programmers are unlikely to benefit from larger regions. Providing RS of full synchronization-free regions may also be unrealistically complex for custom hardware; much proposed hardware support has enforced atomicity of bounded-length regions [3, 31, 33, 42].

RS of statically bounded regions makes software automatically more reliable than under the DRF0 model. In addition to providing SC and thus avoiding difficult-to-understand SC violations, RS of statically bounded regions provides atomicity of its regions, tolerating any atomicity violations that involve statically bounded regions, e.g., common patterns such as improperly synchronized read–modify–write accesses and accesses to multiple variables. As a simple example, an increment $\mathsf{x}{+}{=}5$ that programmers may already expect to be atomic, will execute atomically under RS of statically bounded regions. Similarly, $\mathsf{buffer[pos{+}{+}]} = \mathsf{value}$ accesses two shared variables buffer and pos, and will execute atomically. 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, any assumptions programmers make about the atomicity of source-code-level regions still apply.

Furthermore, the job of various static and dynamic analyses can be simplified by assuming RS of statically bounded regions. Current analyses either assume DRF and (unsoundly) ignore the effects of possible data races, or they consider the effects but incur increased complexity. The model checker CHESS must consider many more possible executions because of the effects of data races [11, 35]. Static dataflow analysis must consider racy interleavings to be fully sound [14]. Various kinds of runtime support either assume DRF unsoundly or add substantial overhead in order to handle data races soundly. The primary performance challenge of software-based multithread record & replay is dealing with the possibility of data races [45]: RecPlay assumes data race freedom unsoundly to reduce costs [40]; Chimera shows that handling data races leads to prohibitively high overhead [27]; other approaces sidestep this problem but incur other disadvantages (e.g., [45]). Kendo unsoundly provides deterministic execution only for race-free programs [37], while CoreDet incurs higher overhead to deal with data races [5]. Dynamic bug detectors must add their own synchronization to deal with the possibility of data races [17, 18].

If RS of statically bounded regions were the default memory model, rather than DRF0, programmers and analyses could assume it, simplifying their jobs. We focus on showing that providing RS of statically bounded regions can have reasonable overhead in commodity systems, making it feasible today for improving software quality and programmer productivity. Future languages and hardware can provide more efficient support for RS of statically bounded regions if there is demand for it.

3. Background: Lightweight Read/Write Locks

This paper presents an approach in which threads gain exclusive or read-shared access to all objects[1] accessed in a statically bounded region, akin to acquiring locks on the objects. Acquiring regular locks on every object would be prohibitively expensive; our approach instead uses an existing optimistic synchronization technique called Octet [9]. This section provides background on Octet; more details appear elsewhere [9].

Octet provides read and write barriers[2] before every access to *potentially shared* objects, which essentially function as lightweight, object-level locks. At run time, each barrier checks that the current thread has exclusive or read-shared access to the object (i.e., holds the "lock"). If so, the access may proceed; otherwise, the barrier must change the object's state (i.e., acquire the "lock") before the access can proceed. Octet barriers provide better performance than traditional locks for three reasons: (1) they support both exclusive and read-shared behavior; (2) there is no explicit release operation, but rather an acquiring thread communicates with thread(s) that have access to the lock in order to gain access; and (3) acquire operations do not need synchronization as long as the thread already has access to the lock.

At run time, each object has a "locality state" that tracks which thread(s) may perform accesses (reads and/or writes) without changing the state and thus without synchronization:

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

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

[2] A read (or write) barrier is instrumentation before every read (write) [46].

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. Octet state transitions fall into three categories that require different levels of synchronization.

- RdSh: Any thread may read but not write the object.[3]

The following pseudocode shows the barriers that the compiler inserts before every program write:

```
if (obj.state != WrEx_T) {
  writeSlowPath(obj); /* change obj.state */
}
obj.f = ... ; // program write
```

and before every program read:

```
if (obj.state != WrEx_T && obj.state != RdEx_T) {
  if (obj.state != RdSh) {
    readSlowPath(obj); /* change obj.state */
  }
  load_fence;
}
... = obj.f; // program read
```

In the *fast path*, each barrier checks whether a thread already has exclusive or read-shared access to the object, based on the object's state. If so, the access may proceed. Otherwise, the barrier executes its *slow path*, which changes the object state. Table 1 shows each possible type of state change. The first three rows (*Same state*) correspond to the fast path and require no synchronization. The last four rows (*Conflicting*) indicate a conflicting access. Before allowing the access to proceed, the barrier must ensure that other thread(s) that currently have *unsynchronized* access to the object do not continue accessing it. The barrier initiates a *coordination protocol* with these other threads to ensure that they stop accessing the object. *Upgrading* transitions are not conflicting, so they require synchronization but not the coordination protocol.

Conflicting transitions and the coordination protocol. A thread that wants to perform a conflicting access is called the *requesting thread* (reqT). The requesting thread performs a coordination protocol with the *responding thread* (respT). For $\mathsf{RdSh} \rightarrow \mathsf{WrEx_{reqT}}$ transitions, every other thread is a responding thread, so reqT performs the coordination protocol with every other thread. here we describe a $\mathsf{WrEx_{respT}} \rightarrow \mathsf{RdEx_{reqT}}$ transition.

Suppose an object o is in the $\mathsf{WrEx_{respT}}$ state, and reqT executes a read barrier before reading o. reqT's barrier executes the slow path, which triggers a conflicting transition to $\mathsf{RdEx_{reqT}}$. reqT first changes o's state to an *intermediate* state $\mathsf{RdEx^{Int}_{reqT}}$ that indicates reqT is in the process of completing a state change to $\mathsf{RdEx_{reqT}}$ and simplifies the protocol by ensuring that only one thread at a time attempts to change o's state. reqT then establishes a roundtrip happens-before relationship with a *safe point* on respT. A safe point is a program point that is definitely not between a barrier and the access it guards. This happens-before relationship ensures that respT "sees" that o is *not* in the $\mathsf{WrEx_{respT}}$ state, triggering the slow path if respT tries to access o again. reqT and respT coordinate *explicitly* if respT is executing normally: reqT sends a request to respT, and respT reaches a safe point and responds (e.g., using per-thread flags and memory fences to establish happens-before relationships). If respT is blocked at a safe point (e.g., waiting to acquire a lock or perform I/O), the threads communicate *implicitly*: reqT synchronizes on respT's blocking flag.

In the explicit protocol, while reqT waits for response(s), it sets its execution state as blocked. This behavior allows *other* requesting threads to perform the implicit protocol with reqT acting as a responding thread. This behavior is important for ensuring progress, as otherwise reqT would be unresponsive to requests from other requesting threads, potentially resulting in deadlock (e.g., suppose two threads try to request objects from each other at the same time). Since reqT allows other requesting threads to perform implicit coordination with it while it performs its own conflicting transition on o, reqT may "lose" exclusive or read-shared access to objects *other than* o during the protocol. This behavior has implications for EnfoRSer's approach, which needs simultaneous access to multiple objects while executing regions.

[3] EnfoRSer does not need nor use Octet's RdSh counter [9]. Instead, EnfoRSer ensures reads to RdSh objects happen after the prior write by issuing a load fence on the RdSh fast path.

4. EnfoRSer: Hybrid Static–Dynamic Analysis

This section describes EnfoRSer, a hybrid static–dynamic analysis that transforms programs to enforce region atomicity at run time.

4.1 Overview

To enforce atomicity of statically bounded regions, 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 (e.g., by adding a header word to each object that tracks the ownership state of the object). Then instrument the region to ensure that a thread holds a lock on each object it accesses before performing its access, and only releases its locks at the end of the region. As this approach implements two-phase locking, it is clear that the execution of regions will be equivalent to a serialized execution of regions.

There are two main problems with this naïve approach. First, the overhead of acquiring locks, which involves atomic operations such as compare-and-swap, on each object access is prohibitive. To reduce this overhead, we use Octet to manage access to objects. By placing an Octet read or write barrier before each access to a potentially-shared object, a region can essentially acquire a lock on an object prior to accessing it. Since Octet only requires expensive synchronization when there may be cross-thread dependences, most object accesses—even accesses to shared objects—can be performed inexpensively without synchronization.

Second, because threads can access objects in arbitrary orders, and locks cannot be released until the end of a region, it seems that regions that acquire multiple locks can deadlock. Interestingly, using Octet barriers as locks cannot result in deadlock: whenever a thread attempts to acquire a lock on an object (by entering an Octet slow path), it may release access to any other object, preventing deadlock. However, this feature merely shifts the problem: if a thread releases a lock on an object accessed during its current region, it loses region atomicity!

To avoid this issue, EnfoRSer's compiler pass transforms regions so that if a thread releases access to an object (we call this a "conflict" since region conflicts always cause one of the threads involved to lose access to an object), it retries the region, and reacquires the necessary locks. The key challenge, then, is to ensure the region can be safely restarted without losing atomicity. We investigate three *atomicity transformations* to address this problem:

1. The *mutual exclusion* transformation hoists Octet barriers into a region "header." If the executing header acquires all locks without conflicts, it executes the rest of the region atomically. If there are conflicts, the header safely restarts since it performs no stores. The challenge of the transformation is determining which locks a region will acquire.
2. The *idempotent* transformation modifies the region to defer stores until all locks have been acquired. If there are conflicts acquiring locks, the region can be safely restarted since it exe-

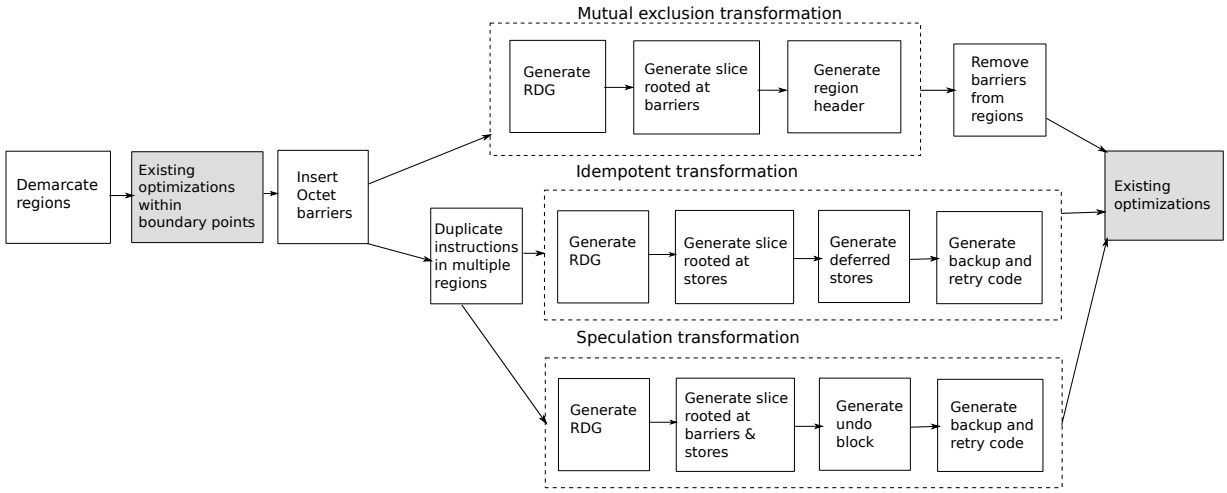


Figure 1. The transformations performed by an EnfoRSer-enabled optimizing compiler. Boxes shaded gray show existing compiler optimizations. Dashed boxes indicate transformations performed once on each region.

cutes idempotently up to that point. The challenge is generating correct code for deferring stores in the face of object aliasing and conditional statements.

3. 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 uses the backed-up values to restore memory to its original state, and restarts the region. The challenges are maintaining the backup variables 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, and any optimizing passes in the compiler are performed, modified to operate within region boundaries. Next, the compiler inserts Octet barriers before object accesses. The compiler then transforms regions using one of the three atomicity transformations. Finally, any remaining compiler optimizations are performed. At run time, a thread executes each region, and if it detects a conflict, it restarts the region safely, preserving atomicity. The atomicity of regions guarantees equivalence to some serialized execution of regions. Section 4.2 describes EnfoRSer's initial instrumentation pass, which demarcates regions and inserts Octet instrumentation. Section 4.3 presents the intermediate representation used by the atomicity transformations. Sections 4.4, 4.5, and 4.6 describe the three atomicity transformations.

4.2 Initial Transformations

This section describes EnfoRSer's initial transformations: splitting a program into regions and inserting Octet instrumentation.

4.2.1 Region Demarcation

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.

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. Appendix A presents an algorithm for demarcating regions.

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 [42], EnfoRSer prohibits inter-region optimizations, by modifying optimization passes that may reorder memory accesses to ensure they do not reorder across boundary points.

4.2.2 Inserting Octet's Lightweight Locks

For all three atomicity transformations, the first step is instrumenting program loads and stores with Octet barriers. Figure 2 shows an example of a region instrumented with Octet barriers. The remainder of this section uses this region as a running example.

EnfoRSer avoids inserting barriers that would be "redundant" because a barrier executing earlier in the region definitely protects the same object. A read barrier is redundant if preceded by a read or write barrier; a write barrier is redundant if preceded by a write barrier. While Octet's default analysis must conservatively assume that any Octet barrier accessing `o` could execute a slow path and "lose" its access to any object *other than* `o` [9], EnfoRSer regions always retry the region in that case, so EnfoRSer uses a more aggressive analysis, limited only by region boundaries.

4.3 The Region Dependence Graph

EnfoRSer's atomicity transformations process each region separately. For each region, they construct a *region dependence graph* (RDG), based on Ferrante et al.'s program dependence graph [16], which captures both data and control dependences in a region. Since regions end at loop back edges, they are acyclic, so the RDG need not consider loop-carried dependences. The RDG also need not track output (write–write) or anti (read–write) dependences.

EnfoRSer's RDG construction algorithm uses a reaching-definitions analysis to compute intraprocedural data dependences in a region. The RDG handles Octet barriers by treating the object reference passed to the barrier as a use of that object. Note that because EnfoRSer performs its analyses at the region level, aliasing relationships that arise due to operations outside the region cannot be inferred. Hence, the construction algorithm adopts a conservative, type-based approach, and 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 illustrates the RDG for the Octet-instrumented region from Figure 2. The dotted lines depict the store–load dependences (which arise between object fields, or between array locations) and def–use dependences (which arise between local variables and references). Note that the construction algorithm's aliasing assumptions lead to a store–load dependence between $\mathsf{s.f} = \mathsf{i}$ and $\mathsf{q} = \mathsf{o.f}$, as it assumes that `s` and `o` may be aliased. The `Entry` node is the root of the RDG and has control dependence edges marked as `T`

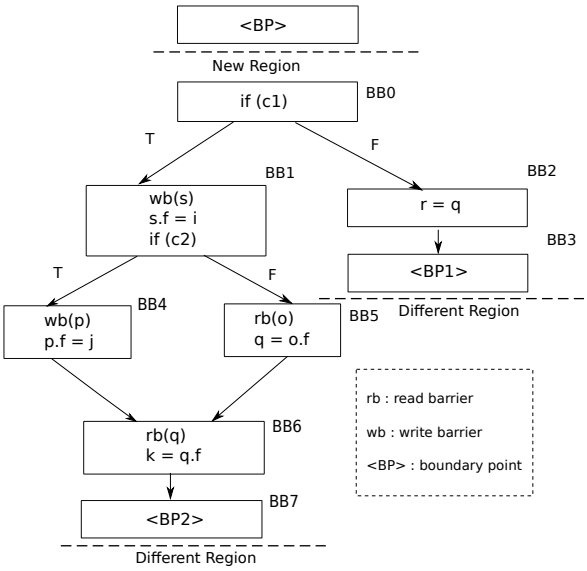


Figure 2. An Octet-instrumented region.

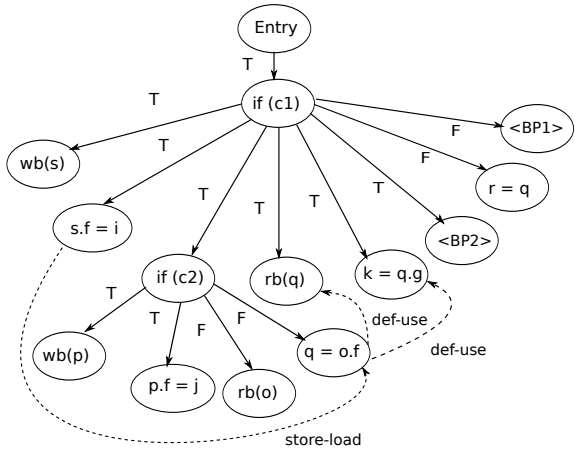


Figure 3. A region dependence graph for the region in Figure 2.

to all the nodes (just 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 in the RDG. Each atomicity transformation must identify which parts of the region are relevant for its transformations. They perform static program slicing of the RDG (based on static program slicing of the program dependence graph [23]) 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.

4.4 Enforcing Mutual Exclusion of Conflicting Regions

The first atomicity transformation supported by EnfoRSer enforces mutual exclusion of potentially conflicting regions. Each transformed region acquires all of its "locks" (i.e., executes all of its Octet barriers) at the beginning of the region—we call this part of the region's *header*—prior to performing any instructions with side effects (program stores), allowing restarting of the header in case atomicity might have been lost. When the rest of the region executes, all needed locks will be held, ensuring atomicity as in two-phase locking.

At a high level, constructing the region's header involves (i) computing a program slice that is the union of backward slices from

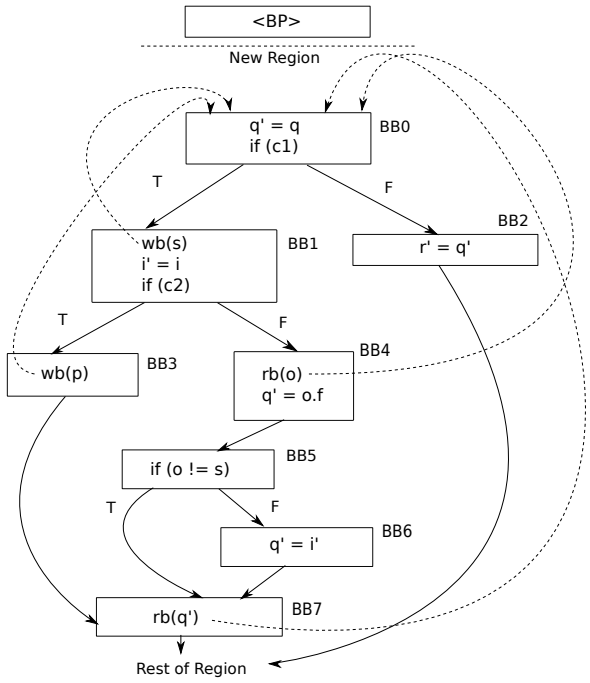


Figure 4. Mutual exclusion header generated for Figure 2. Dotted arrows indicate branches taken if a barrier's fast-path check fails.

every Octet barrier in a region; (ii) generating code from the slice so it can correctly and safely execute the necessary Octet barriers without side effects (i.e., without stores); and (iii) generating code to retry the header if an Octet barrier fails its fast-path check and takes a slow path. The primary challenge is the second step, generating code for the region header, which we explain next. Figure 4 shows the header generated for the code in Figure 2.

4.4.1 Generating the Region's Header

The algorithm begins by computing a backward slice on the RDG from all Octet barriers in the region. Figure 5 shows the slice on the RDG from Figure 3 rooted at Octet barriers. Executing the instructions in this slice will perform exactly the operations needed to acquire all of the locks in the region (i.e., execute all of the barriers). If the slice contains no side effects (no stores), then code generation is straightforward: the algorithm generates code from the slice and adds code to retry the header from the start after any Octet barrier's slow path executes (since an object previously acquired by the region may have been lost).

This simple approach is complicated by the fact that the header slice often *does* contain stores. In particular, the object that is the target of an Octet barrier may be determined by a load, which in turn may be dependent on a store. Consider BB6's read barrier rb(q) in Figure 2. The object q may have been loaded from o.f in BB5, which may, in turn, have been stored to in BB1 when executing s.f = i. As a result, s.f = i appears in the slice of Figure 5. Executing the store as part of the region's header would violate the requirement that headers be side-effect free.

To avoid this issue, the header generation algorithm replaces all stores in the header with assignments to new local variables. The key is ensuring that any loads that may be dependent on the rewritten store correctly get their values from that local variable, if necessary. Two factors preclude simply rewriting loads to read from the newly-created local variable: (i) the uncertainty of aliasing means that the store may not affect the load (e.g., the load of o.f is dependent on the store to s.f only if o and s are aliased); and (ii) if a store executes conditionally, the load should not get its value from a local variable, but rather must load from the original object. We address these cases in turn.

Store–load aliasing. The RDG in Figure 5 includes a store–load edge from s.f = i to q = o.f because RDG construction conser-

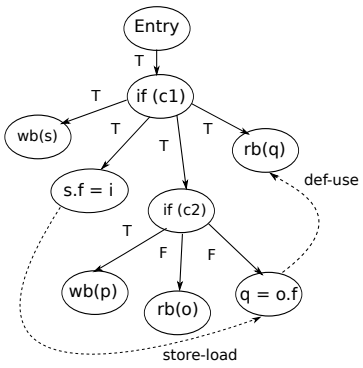


Figure 5. A slice of Octet barriers from the RDG in Figure 3.

vatively assumes all type-compatible accesses might alias. As described above, header generation substitutes a definition of a new temporary variable, i' = i in place of the store s.f = i, as shown in BB1 of Figure 4. 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 algorithm emits a series of alias checks to disambiguate between the scenarios (one for each potentially aliased store that may reach the load). The algorithm first emits the original load. Then, for each store on which the load is dependent, the algorithm generates an assignment from the corresponding local variable, guarded by an alias check. By performing these checks in program order, the load will ultimately produce the correct value, even if multiple aliasing tests pass. A similar situation can arise with loads and stores of array elements. In this case, the aliasing checks must not only make sure that the array objects are the same, but also that the index expressions evaluate to the same value.

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.

Conditional dependences. If a store does not dominate its dependent load, just performing an alias check is not enough, as the store *might not happen* during execution, and hence the load should not read from the store's local variable. To handle this case, for each store that does not dominate the load, the header generation algorithm reproduces all conditionals up to the most recent common control ancestor of the load and store, and re-executes them prior to the alias check and load. Hence, the load will only depend on the store if the alias check passes *and* the store actually executed.

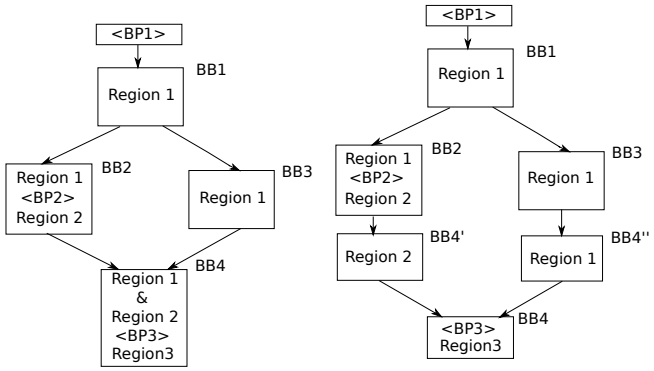
Our running example in Figure 2 does *not* show such a case, but suppose s.f = i executed only under condition c3. Then prior to BB5 in Figure 4, header generation would insert code to perform the alias check only if c3 were true.

Variable renaming in the header. To ensure that the header does not overwrite local variables in the region, all definitions and uses of local variables are changed to use newly-allocated local variables. So that any renamed uses see the proper values, any variables that are live-in to the region are copied into their renamed counterparts at the beginning of the header. The assignment q' = q in BB0 shown in Figure 4 is an example of this transformation.

A downside of the mutual exclusion transformation is that it generates redundant code: much of the header's work to determine which barriers to acquire is repeated by the rest of the region. A promising avenue for optimization is to eliminate these redundancies. The rest of the region can reuse results computed in the header, essentially a form of common subexpression elimination. A challenge is that all headers that reach an instruction must produce an operation's result in order to remove the operation from the rest of the region.

4.5 Enforcing Atomicity with Idempotence

The *idempotent* transformation defers side effects (program stores) until the end of the region. The rest of the region, which includes all the barriers, executes without side effects, *idempotently*, so it



(a) Overlapping regions (b) After duplicating instructions

Figure 6. An example showing how instructions are duplicated so that each instruction is statically in a single region.

can be restarted if a barrier detects a possible region conflict. While the mutual exclusion transformation essentially "hoists barriers up" into a region header, the idempotent transformation "pushes stores down" to the end of the region.

4.5.1 Initial Transformations

We first describe transformations performed before the idempotent transformation. These transformations are also performed before the speculation approach (Section 4.6).

Statically unique regions. In the idempotent approach, a barrier handles potential conflicts by returning control to the top of the region. However, if the barrier statically appears in multiple regions, the appropriate region start point to return to cannot be determined at compile time. In Figure 6(a), any instructions in BB4 before the boundary point (<BP3>) are part of two regions. The idempotent (and speculation) transformations first transform a method by duplicating instructions and control flow as necessary, so that every static instruction is in its own region (note that boundary points are not considered part of any region). First, a simple data flow analysis determines which boundary point(s) reach each instruction (i.e., which region(s) each instruction resides in). Second, in topological order starting at the 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 6(b) shows the result of this transformation, which has split instructions above BB4's boundary point into two separate basic blocks, BB4' and BB4''.

We note that duplicating a region does not include duplicating any region boundary points that terminate the region, so the number of region boundary points remains fixed, and duplication cannot result in an exponential blowup in code size.

Backing up local variables. Although the idempotent transformation defers stores, the region can also have side effects by modifying local variables, which 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.

4.5.2 Idempotent Transformations

To defer program stores, the transformation replaces each static store with a definition of a new local variable. 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 set of stores that actually executed.

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. Handling these loads is exactly the same problem as dealing with

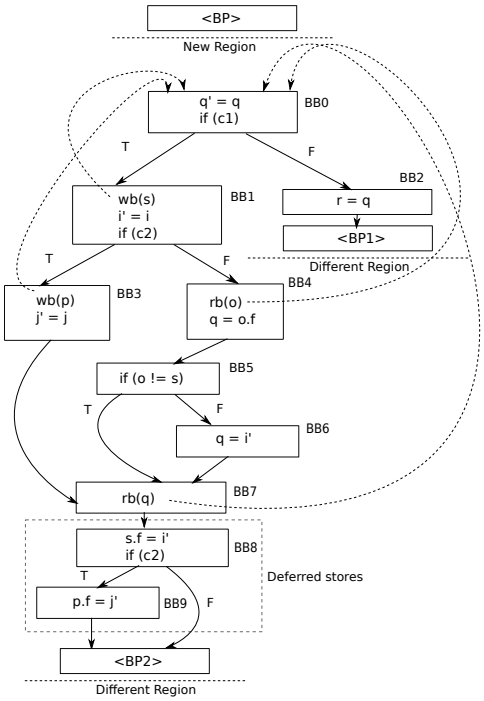


Figure 7. The region from Figure 2 after EnfoRSer's idempotent transformation.

store–load dependences in the mutual exclusion approach, and the transformation uses the same code generation strategy.

Deferred stores. It is nontrivial to generate code at the end of the region to perform the region's stores because the code should execute 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. The transformation generates each deferred store using its 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; these conditions are re-checked before performing any deferred store.

The transformation backs up any variable used in a conditional expression that may be overwritten during the region; these backups are used in the conditionals guarding deferred stores, ensuring that the conditionals resolve exactly as they did in the original region. Similarly, any store to an object field or array in the original region is dependent on the base address of the object or array. If these base addresses 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.

Retrying the region. The transformation inserts code after each Octet slow path that (1) restores backed-up local variables and (2) redirects control flow to the start of the region.

Example. Figure 7 shows the region from Figure 2 after the idempotent transformation. The transformed region defers the stores in BB1 and BB3 by writing to new local variables i' and j'. BB4–BB6 handle a load from a potentially store–load dependence: the original load is followed by an assignment from the deferred store's local variable that is guarded by an alias check. In BB8 and BB9, deferred stores are finally performed using each store's backup local variable, guarded by conditionals so that only the correct stores execute. After each slow path, the code restores the value of q from q' (not shown) and control returns to the region start (dashed arrow indicates the edge is taken only if the barrier's slow path executes).

4.6 Enforcing Atomicity with Speculation

EnfoRSer's third atomicity transformation enables *speculative execution*. Rather than transforming a region so that all locks are acquired (i.e., all Octet barriers are executed) before any stores are performed (as in the mutual exclusion and idempotent approaches), this transformation leaves the order of operations in the region unchanged, but perform stores speculatively: before a store executes, instrumentation saves the old value of the field (or array element) being updated. If an Octet barrier detects a possible region conflict, the region maintains atomicity by "rolling back" stores the region has performed, using the saved values. The primary challenge is generating code that correctly backs up stored-to variables and rolls back speculative state if a barrier detects a possible region conflict.

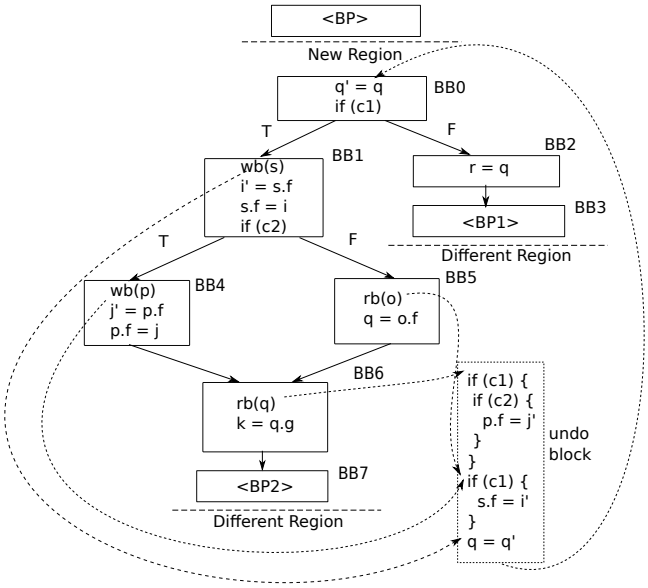


Figure 8. The region from Figure 2 after being transformed by EnfoRSer's speculation transformation.

Since, like the idempotent transformation, the speculation transformation retries conflicting regions, the transformation first ensures each instruction is in its own region and backs up live-in local variables that might be defined in the region (Section 4.5.1).

Supporting speculative stores. Since regions are acyclic, every static store executes at most once per region execution. Hence, the speculation approach's transformation can associate each store with its own 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 these problems, 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 the store up through its control ancestors. As in the idempotent approach, conditional variables and object base addresses are backed up if necessary during regular execution. These backups are used during rollback.

For each Octet barrier in the region, control flow is generated to jump to the appropriate location in the undo block if the slow path is taken. For each barrier b, the jump target in the undo block is the first undo operation associated with a store that is a predecessor of b in the region.

Figure 8 shows the result of transforming the region in Figure 2 to support speculative execution. If a barrier's fast-path check fails, control jumps into the undo block; the figure shows these conditional jumps with dashed lines. The undo block rolls back the stores in reverse topological order, with each store performed if and only if conditionals indicate it executed in the original code. Since the Octet slow path allows other threads to acquire objects that the undo

block will write to, the generated code defers executing the slow path until after the undo block executes (not shown).

5. Implementation

We have implemented EnfoRSer in Jikes RVM 3.1.3, a high-performance Java virtual machine [4] that provides performance competitive with commercial JVMs.[4] We build our implementation on top of the publicly available implementation of Octet.[5]

Modifying the compilers. Jikes RVM uses two just-in-time dynamic compilers that perform method-based compilation, which we modify to enforce RS in both the application and Java library methods. The first time a method executes, the *baseline* compiler compiles it directly from Java bytecode to machine code. As a method becomes hot, the *optimizing* compiler recompiles it at successively higher levels of optimization.

By design, execution spends most of its time executing optimized code. The optimizing compiler uses an intermediate representation (IR) with three levels: a high-level IR called *HIR* that resembles bytecode but with virtual registers; a lower-level IR called *LIR* that resembles medium-level IR using three-address code in traditional compilers; and a machine-specific IR (our implementation targets IA-32). We modify the optimizing compiler so it performs the atomicity transformations after HIR optimizations and before LIR optimizations, allowing some optimizations to occur in the absence of Octet barriers. LIR optimizations occur afterwards, and can help clean up EnfoRSer's generated code.

Since the baseline compiler does not use an IR, it would be hard to perform atomicity transformations. Instead, our prototype implementation simulates the cost of enforcing atomicity in baseline-compiled code by inserting (1) Octet barriers, (2) instrumentation that logs each store in a memory-based undo log (not in local variables), and (3) instrumentation that resets the undo log pointer at each boundary point. All EnfoRSer configurations use this instrumentation, which is unsound since it does not support rollback. Since rollbacks are infrequent and execution spends a small fraction of time in baseline-compiled code, we expect performance to closely match what a fully sound implementation would achieve.

The compiler performs EnfoRSer's transformations on application and library code. Since Jikes RVM is written in Java, VM code can get 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 some methods from instrumentation by EnfoRSer; instead, the compiler adds only Octet instrumentation to these methods. We exclude 21 methods across all benchmarks and 10 methods in the Java libraries. For one benchmark (jython9), we prevent the mutual exclusion approach from instrumenting java.util.concurrent.locks.* methods when they get inlined into application code, since we have not completely narrowed down the problem.

Demarcating regions. EnfoRSer demarcates regions at synchronization operations (lock acquire–release, wait–notify, thread fork–join), method calls, and loop back edges. Conveniently, these are essentially the same program points that are GC-safe points in Jikes RVM and other VMs. Safe points cannot be included in our regions anyway since that would violate atomicity.

Object allocations are also boundary points in our implementation, since they are safe points. A production implementation could either define regions as being bounded at allocation (non-array allocations include a constructor call in any case), or memory allocation could be speculatively hoisted above regions. Jikes RVM makes each loop header (rather than loop back edge) a safe point, so in our implementation regions are bounded at loop headers. To simplify the implementation, we currently bound regions along special control-flow edges for (1) switch statements and (2) run-time checks for guarded inlining.

Exception-handling edges exit the current region. The next region starts at a catch block in the current method or in some caller method on the stack. For exceptions handled by a catch block in the current method, the exception-handling edge is captured by the RDG. For other exceptions, the region exits early along control flow not captured by the RDG, which can affect correctness for the mutual exclusion and idempotent approaches since they reorder loads and stores. In Appendix B, we discuss but do not implement a deoptimization-based approach to handle these exceptions.

The optimizing compiler performs some optimizations—method inlining and loop unrolling—that may remove safe points that existed in the unoptimized code. As a result, the regions identified by the optimizing compiler may be larger than those that appear at the source-code level. However, these regions are *strictly larger* than the original regions, so any atomicity expectations the programmer might have based on the source code will still hold.

Preserving RS in the compiler. We prevent the compiler from reordering across boundary points before it performs EnfoRSer's transformations by identifying optimizations that reorder across boundary points. (Alternatively, we could insert soft fences [33] at boundary points.) Before EnfoRSer's transformation (i.e., during HIR optimizations), Jikes RVM performs just one optimization that can reorder loads or stores across boundary points: local common subexpression elimination (CSE). We disable reordering of loads and stores across boundary points but find that the performance effect is negligible. In contrast, Marino et al. measure a 5% average slowdown from disallowing cross-region reordering [33]. Our results differ from theirs in two keys ways. First, Jikes RVM's optimizing compiler must already be conservative about performing optimizations across region boundaries since they are also safe points where GC can occur. Second, the Jikes RVM optimizing compiler is not performing optimizations as advanced as the LLVM compiler [26] used in the DRFx work: Jikes RVM's optimizing compiler is less aggressive since it is a just-in-time compiler, and some of Jikes RVM's most aggressive optimizations (O3) are disabled in version 3.1.3 since they are not robust.

After EnfoRSer's transformations, reordering *within* regions is still permitted—particularly for the mutual exclusion and idempotent approaches since they move barriers or stores. Reordering *across* boundary points is automatically prevented since Octet barriers contain conditional fences. A compiler can perform speculative optimizations around Octet barriers to reorder only on the fast path but not the (fenced) slow path, with fix-up code on the slow path to preserve RS, similar to prior work's optimization [34].

6. Evaluation

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

6.1 Methodology

Benchmarks. The experiments execute our modified Jikes RVM on the multithreaded DaCapo benchmarks [6] versions 2006-10-MR2 and 9.12-bach with the large workload size (excluding benchmarks that Jikes RVM cannot run) and fixed-workload versions of SPECjbb2000 and SPECjbb2005 [6].[6]

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

Experiments run on an AMD Opteron 6272 system with 4 16-core processors running Linux 2.6.32. Experiments run on only 32 cores (using the Linux taskset command) due to an anomalous result with 64 cores where Octet and EnfoRSer actually outperform the baseline on at least one benchmark. (We find that adding *any* kind of instrumentation can *improve* performance, apparently because it dampens an oversaturation issue on the NUMA platform.)

[4] http://dacapo.anu.edu.au/regression/perf/9.12-bach.html

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

[6] http://www.spec.org/jbb2000, http://www.spec.org/jbb2005

	Insts. executed	Insts. / region	Dynamic regions retried Mut. excl.	Idemp.	Spec.
eclipse6	7.3×10^{10}	24	<0.01%	<0.01%	<0.01%
hsqldb6	5.0×10^{9}	20	0.50%	0.50%	0.54%
lusearch6	5.8×10^{9}	25	<0.01%	<0.01%	<0.01%
xalan6	9.4×10^{10}	24	0.76%	0.80%	0.84%
avrora9	3.8×10^{10}	22	0.52%	0.52%	0.52%
jython9	7.7×10^{10}	23	<0.01%	<0.01%	<0.01%
luindex9	3.6×10^{9}	23	<0.01%	<0.01%	<0.01%
lusearch9	2.4×10^{9}	24	<0.01%	<0.01%	<0.01%
pmd9	1.4×10^{9}	21	0.07%	0.06%	0.11%
sunflow9	4.7×10^{9}	26	<0.01%	<0.01%	<0.01%
xalan9	4.7×10^{10}	25	1.06%	1.06%	1.06%
pjbb2000	2.5×10^{9}	18	1.21%	1.21%	1.21%
pjbb2005	1.6×10^{10}	21	3.00%	3.29%	3.50%

Table 2. Characteristics of executing regions. The first two columns are total dynamic instructions and average dynamic instructions executed per region. The last three columns are the percentage of dynamic regions that retry for each of the three atomicity approaches. The benchmark name suffixes 6 and 9 distinguish the 2006 and 2009 DaCapo benchmarks. pjbb200{0,5} are fixed-workload versions of SPECjbb200{0,5}.

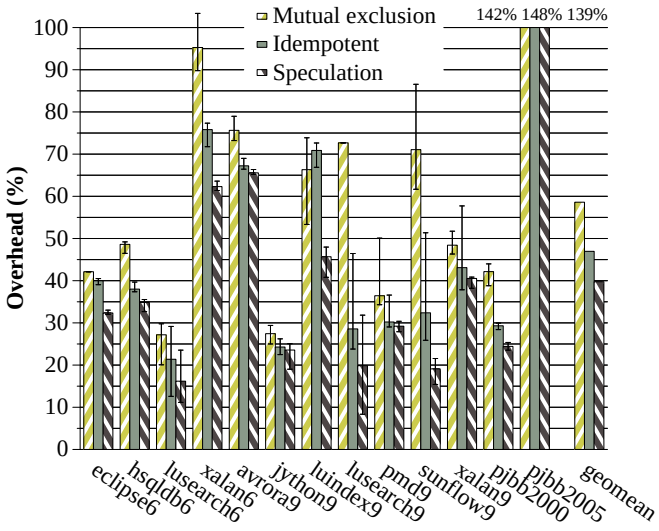


Figure 9. Run-time overhead over an unmodified JVM of providing RS with EnfoRSer's three atomicity transformations.

6.2 Run-Time Characteristics

Table 2 reports characteristics of executing regions averaged across 10 trials. The first two columns, measured without any EnfoRSer instrumentation, show that the programs execute billions of instructions, divided into regions that execute 18–26 IA-32 instructions each on average. Across all programs, each region executes 23 instructions on average. Our region sizes are on the same order as DRFx's statically bounded regions, which appear (from graphs) to be about 10 instructions per region [33].

The last three columns show that the vast majority of regions do not detect a conflict,[7] suggesting optimistic locking is worthwhile and that livelock is unlikely. We note that since EnfoRSer instruments all accesses—not just accesses to shared data—and many regions access only thread-local data, even $\geq$0.1% of all accesses can be a significant amount of communication [9].

6.3 Performance

Figure 9 shows the overhead that providing EnfoRSer-based RS adds over unmodified Jikes RVM. Each bar is the median of 10 trials (to minimize effects of machine noise), and the intervals are 95% confidence intervals centered at the mean.

All approaches use Octet, which alone adds 26% overhead on average (not shown). The idempotent and speculation approaches duplicate instructions that are statically in multiple regions, which adds 7% alone (not shown). The three approaches, mutual exclusion, idempotent, and speculation, add 59%, 47%, and 40% total overhead, respectively. The higher overhead of the mutual exclusion approach is from headers duplicating much of their regions' logic. The idempotent approach avoids this duplication but still incurs costs to buffer and replay stores correctly and to check for aliasing at loads. Speculation is the best-performing approach since it adds the least complexity, requiring only backing up each store's old value to a local variable. In a more aggressive compiler, the other approaches might have more of an advantage by enabling more intra-region reordering.

These results show that EnfoRSer enforces RS at a reasonable overhead of 40% (using the speculation approach). To our knowledge, this represents the lowest reported overhead of RS enforcement on commodity systems. The best-performing prior work requires additional cores to be available to avoid adding over 100% overhead [38].

EnfoRSer adds the highest overhead for pjbb2005 (139–148%). This overhead primarily comes from Octet, which alone adds 111% to pjbb2005 due to a relatively high conflicting transition rate. Future work could consider a less-aggressive concurrency control for highly conflicting objects or regions. To measure the cost of retrying regions, we use configurations that (unsoundly) do not retry after slow paths. We find that the cost of retrying is negligible (not shown), which is unsurprising given the low retry rate (Table 2).

The idempotent and speculation approaches are related to software transational memory (STM) [21, 22]. We evaluate an alternative version of the speculation approach that uses an in-memory buffer for the undo log. We find that this approach adds 61% total overhead (not shown), significantly more than EnfoRSer's speculation approach. Furthermore, STMs incur additional costs to (1) maintain read/write sets, which EnfoRSer avoids by conservatively assuming all conflicting accesses are regions conflicts—a reasonable tradeoff for short regions—and (2) perform concurrency control, for which EnfoRSer uses Octet.

Compilation time. The overheads in Figure 9 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. We have focused on correctness over performance in our prototype implementation's compiler passes, so we expect that compile-time overhead could be substantially lowered with more work. We find that compile time increases over unmodified Jikes RVM by 3.2X, 2.5X, and 2.2X for the mutual exclusion, idempotent, and speculation approaches, respectively. Since dynamic compilation occurs at run time, it can slow overall execution time by competing with the application for execution resources and by causing code to be less optimized overall as part of Jikes RVM's adaptive compilation strategy. By using a methodology that separates out the costs of compilation and application run time [19], we have verified that the effect of EnfoRSer's compilation overhead is modest (a few percent, relative to baseline execution time).

Sequential consistency. While RS of statically bounded regions provides many benefits, one immediate benefit is that an RS execution is also sequentially consistent (SC). As a result, EnfoRSer can be used to provide end-to-end SC, and do so without the whole-program static analysis or custom hardware required by other approaches (Section 7). We compare EnfoRSer to providing end-to-end SC with memory fences, as that is the alternative approach that works on commodity hardware. On IA-32's total store order (TSO) memory model, end-to-end SC can be achieved with a full mem-

[7] Although in theory the atomicity approaches should retry identically, we find that some differences across approaches are statistically significant. We believe differences are due to nondeterministic optimized compilation outcomes (since the approaches have different compilation overheads).

ory fence after each store and a soft fence[8] after each load, which we find slows programs by 46% on average. On weaker hardware memory models, end-to-end SC requires a memory fence after each load and store, which slows programs by slows programs by 206% on average. In contrast, EnfoRSer provides a stronger property, RS, with lower overhead, and EnfoRSer should perform well on weaker hardware memory models.

7. Related Work

Section 2 covered the closest related work on enforcing RS in software. This section covers other approaches.

Enforcing atomicity. Prior work introduces custom hardware support to enforce atomicity of regions. *Atom-Aid* tolerates some atomicity violations [31], while *BulkCompiler* focuses on providing SC [3]. 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.

Transactional memory. Transactional memory (TM) guarantees atomicity of programmer-annotated regions [21, 22], so it could be used to provide RS. Hammond et al. introduce a memory model and associated hardware where all code is in transactions, providing RS [20]. However, efficient software-only TM support remains elusive. Software TM requires read/write sets and memory-based logging—costs that EnfoRSer avoids by using statically bounded regions and conservatively retrying on potential conflicts.

Checking for region conflicts. Prior work *checks* region conflict freedom (RCF) of synchronization-free regions in order to avoid SC violations or detect data races [15, 30, 33, 42]. Regions can be delimited only by synchronization operations [30] or be statically bounded to simplify the hardware [33, 42], or extended somewhat beyond synchronization-free regions [15]. These approaches rely on custom hardware [30, 33, 42] or add significant overhead [15]. 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.

Sequential consistency. Providing end-to-end SC requires restricting reordering by both the compiler and hardware. Whole-program static analysis called *delay sets* detects possible SC violations and introduces memory fences conservatively (e.g., [41, 44]). *Delay set* analysis fundamentally needs all of the code to perform a whole-program analysis. EnfoRSer is an intraprocedural analysis so it can instrument each method as it is compiled without needing to consider other methods.

Hardware can speculatively reorder loads and stores (e.g., [29, 39]). Compilers can be modified to enforce SC, with additional hardware support providing end-to-end SC [34, 43]. Providing end-to-end SC—a weaker property than RS—seems destined to add nontrivial overhead to restrict compiler and hardware reordering.

Race detection. EnfoRSer could avoid detecting conflicts for accesses definitely not involved in data races. However, static race detection requires whole-program analysis that provides limited benefit to dynamic analysis [27], while dynamic data race detection slows programs by several times [17].

8. Conclusions

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

Acknowledgments

Thanks to Meisam Fathi Salmi, Jipeng Huang, Todd Millstein, Madan Musuvathi, Satish Narayanasamy, and Chris Stone for valuable discussions.

[8] A soft fence restricts compiler reordering but is a no-op in the final machine code [33].

Algorithm 1 Analysis for demarcating regions

```
input: startPoints                          # first instruction of method
regions ← {}                                # set of regions
while startPoints ≠ ∅ do
  region ← {}
  regionInsts ← startPoints.pop()
  while regionInsts ≠ ∅ do
    i ← regionInsts.pop()
    if i is a boundary point then
      startPoints ← startPoints ∪ i
    else
      region ← region ∪ i
      regionInsts ← regionInsts ∪ {successors of i}
    end if
  end while
  regions ← regions ∪ region
end while
return regions
```

A. Demarcating Regions

Algorithm 1 presents the algorithm for demarcating regions at boundary points. The compiler transformations for all three approaches each first call this algorithm. The algorithm processes a worklist of region start points called *startPoints*, initialized with the method's first instruction in the compiler's intermediate representation (IR). The algorithm keeps adding reachable instructions to the current region until it encounters boundary points, which it processes as region start points. Once all non-boundary points have been added to the region, the region is complete, and the algorithm adds it to the set of regions.

B. Exceptions in Transformed Regions

The implementation demarcates regions at exception-handling edges, i.e., a `catch` block starts a new region. Exceptions present a problem for the mutual exclusion and idempotent transformations. Transformed regions can throw exceptions (e.g., a null pointer exception trying to load from an object field thrown from a header in the mutual exclusion approach or the region in the idempotent approach). Simply letting execution flow to the catch block is incorrect: in the original code, some stores could have executed before the exception, while in the transformed code, no stores have executed. Exceptions are not a problem for the speculation approach: a thrown exception exits the region, preserving program semantics and region atomicity.

The mutual exclusion and idempotent approaches could handle runtime exceptions by performing just-in-time *deoptimization*. On an exception thrown from a header, the runtime system would transfer control to a deoptimized version of the method, e.g., a baseline-compiled version. By continuing execution at the same region boundary point, the exception would be thrown correctly, preserving semantics and RS. The idempotent approach would also need to restore backed-up local variables.

Table 3 captures the scope of this problem. It shows how many `RuntimeException` instances are thrown from EnfoRSer-instrumented application and library methods that are compiled by the optimizing compiler. (Exceptions other than `RuntimeException` must be thrown with Java's `throw` keyword, which is a safe point and hence outside of regions.) These exceptions could occur inside of the transformed regions and then possibly violate program semantics. We have not encountered any problems in practice, Overall, the programs throw few (dynamic and distinct) exceptions, so deoptimization would have reasonable performance.

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] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. BulkCompiler: High-performance Sequential Consistency through

	Dynamic	Distinct
eclipse6	58,000	1
hsqldb6	0	0
lusearch6	66	2
xalan6	0	0
avrora9	0	0
jython9	0	0
luindex9	0	0
lusearch9	0	0
pmd9	0	0
sunflow9	0	0
xalan9	0	0
pjbb2000	0	0
pjbb2005	0	0

Table 3. Total dynamic and distinct run-time exceptions in optimized code, rounded to two significant digits.

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

[4] 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.

[5] 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.

[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, 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.

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

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

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

[13] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring Locks for Atomic Sections. In *PLDI*, pages 304–315, 2008.

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

[15] 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.

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

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

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

[19] A. Georges, L. Eeckhout, and D. Buytaert. Java Performance Evaluation through Rigorous Replay Compilation. In *OOPSLA*, pages 367–384, 2008.

[20] 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.

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

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

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

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

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

[26] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, pages 75–88, 2004.

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

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

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

[30] 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.

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

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

[33] 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.

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

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

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

[37] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, pages 97–108, 2009.

[38] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. ...and region serializability for all, 2013.

[39] 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.

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

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

[42] 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.

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

[44] 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.

[45] 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.

[46] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers Reconsidered, Friendlier Still! In *ISMM*, pages 37–48, 2012.