Toward Efficient Strong Memory Model Support for the Java Platform via Hybrid Synchronization [*]

Aritra Sengupta Man Cao Michael D. Bond
Ohio State University
{sengupta,caoma,mikebond}@cse.ohio-state.edu

Milind Kulkarni
Purdue University
milind@purdue.edu

Abstract

The Java memory model provides strong behavior guarantees for data-race-free executions. However, it provides very weak guarantees for racy executions, leading to unexpected, unintuitive behaviors. This paper focuses on how to provide a memory model, called *statically bounded region serializability* (SBRS), that is substantially stronger than the Java memory model. Our prior work introduces SBRS, as well as compiler and runtime support for enforcing SBRS called *EnfoRSer*. EnfoRSer modifies the dynamic compiler to insert instrumentation to acquire a lock on each object accessed by the program. For most programs, EnfoRSer's primary run-time cost is executing this instrumentation at essentially every memory access.

This paper focuses on reducing the run-time overhead of enforcing SBRS by avoiding instrumentation at every memory access that acquires a per-object lock. We experiment with an alternative approach for providing SBRS that instead acquires a single *static* lock before each executed region; all regions that potentially race with each other—according to a sound whole-program static analysis—must acquire the same lock. This approach slows most programs dramatically by needlessly serializing regions that do not actually conflict with each other. We thus introduce a *hybrid* approach that judiciously combines the two locking strategies, using a cost model and run-time profiling.

Our implementation and evaluation in a Java virtual machine use offline profiling and recompilation, thus demonstrating the potential of the approach without incurring online profiling costs. The results show that although the overall performance benefit is modest, our hybrid approach never significantly worsens performance, and for two programs, it significantly outperforms both approaches that each use only one kind of locking. These results demonstrate the potential of a technique based on combining synchronization mechanisms to provide a strong end-to-end memory model for Java and other JVM languages.

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

PPPJ '15, September 8–11, 2015, Melbourne, FL, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3712-0/15/09…$15.00.
http://dx.doi.org/10.1145/2807426.2807432

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; run-time profiling

1. Introduction

It is challenging to develop, debug, and test multithreaded programs largely because of the many possible program behaviors that a concurrent execution can have. In particular, modern language memory models—including the Java memory model and thus the memory model for other Java platform languages—provide very weak guarantees for executions with data races (Section 2.1) [2, 3, 9, 25, 35]. This situation represents a serious impediment to reasoning about program behavior and providing strong semantic guarantees.

Much prior work has focused on guaranteeing *sequential consistency* (SC) [21]. Although SC is stronger than the Java memory model, it fails to match programmers' expectations and thus does not eliminate many unexpected behaviors [2]. Furthermore, enforcing *end-to-end*[1] SC has proven expensive (Section 2.1).

In prior work, we introduced a memory model called *statically bounded region serializability* (SBRS) that guarantees serializability of regions that are intraprocedural, acyclic, and synchronization free (Section 2.1) [31]. Our prior work introduces an enforcement mechanism for SBRS called *EnfoRSer* that transforms the compiled program so that it acquires a *per-object lock* at each memory access (Section 2.2) [31]. A difficult-to-avoid cost of EnfoRSer is that instrumentation must acquire a lock at most memory accesses. Furthermore, EnfoRSer modifies the compiler to transform statically bounded regions to execute either idempotently or speculatively; the transformed code adds run-time overhead (Section 2.2). In this paper, we refer to EnfoRSer's per-object locks as *dynamic locks* because they are associated with run-time objects, and we refer to the version of EnfoRSer that uses dynamic locks as *EnfoRSer-D*.

This paper's goal is to provide SBRS with lower overhead than prior work achieves. The high-level direction is to reduce the instrumentation cost for regions that conflict rarely, if ever. The mechanism for reducing instrumentation is to enforce atomicity using coarse-grained locking at the region level, instead of the object level. In order for a region to ensure atomicity by acquiring a lock, the approach must ensure that two regions acquire the same lock if they might race with each other (i.e., if the regions each have one of a pair of potentially racy accesses). We refer to these locks as *static locks* because they are associated with static program sites. A *site* is a static memory access; it is identified uniquely by a method and a bytecode index. We refer to EnfoRSer that uses static locks

[1] Enforcing a memory model *end-to-end* means the system enforces the memory model with respect to the original program. That is, both the compiler and hardware must respect the memory model.

to guard regions as *EnfoRSer-S*. EnfoRSer-S's approach often leads to over-synchronization that harms scalability significantly. Static locks thus must be applied judiciously: they should be applied to a region only when they will not incur significant unnecessary contention; other regions should use dynamic locks.

We thus introduce a *hybrid* version of EnfoRSer called *EnfoRSer-H* that selectively applies static and dynamic locks. EnfoRSer-H makes use of an *assignment algorithm* that chooses, for each site, whether to use static or dynamic locks, subject to the constraint that all sites that race with each other must use the same kind of locking. The assignment algorithm makes its decisions based on a cost model and run-time profiling information.

We have implemented EnfoRSer-S and EnfoRSer-H in Jikes RVM, a high-performance Java virtual machine (JVM) [5, 6], on top of our existing EnfoRSer-D implementation [31]. To avoid the engineering challenge of invalidating and recompiling all methods affected by run-time locking changes, our evaluation instead uses a methodology that runs two iterations of the application: the first iteration collects profile information using static locks only; then the JVM recompiles all methods based on the assignment algorithm; and finally the second iteration executes using this new locking scheme.

Our evaluation compares the run-time characteristics and performance of EnfoRSer-D, EnfoRSer-S, and EnfoRSer-H on benchmarked versions of large, real, multithreaded applications [8]. EnfoRSer-D's use of dynamic locks minimizes contention but incurs lock acquire overhead at each potentially racy memory access. On the other hand, EnfoRSer-S's exclusive use of static locks reduces the number of lock acquire operations but leads to high contention and thus substantial run-time slowdowns. By hybridizing dynamic and static locks, EnfoRSer-H is able to get the benefits of both approaches. Most programs cannot benefit much from static locks, so EnfoRSer-H performs almost identically to EnfoRSer-D; notably, EnfoRSer-H does not harm performance in these cases but instead correctly chooses to use dynamic locks. Since most programs fall into this category, EnfoRSer-H's overall improvement over EnfoRSer-D is modest: EnfoRSer-H incurs 26% run-time overhead on average, compared with 27% for EnfoRSer-D. However, for a few programs that can benefit from static locks, EnfoRSer-H's selective use of static locks leads to substantially lower run-time overhead than EnfoRSer-D. Overall, EnfoRSer-H demonstrates opportunities for exploiting low-contention parts of applications to provide a strong memory model, SBRS, with lower overhead than prior work.

2. Background and Motivation

This section first describes existing memory models, in order of increasing strength, concluding with the statically bounded region serializability (SBRS) memory model. Section 2.2 describes our prior work's compiler- and runtime-based approach for enforcing SBRS and motivates its performance limitations and opportunities for improvement.

2.1 Memory Models

A *memory model* defines the possible values for a load from shared memory [2]. This paper is concerned with language memory models that must be enforced *end-to-end*, i.e., must be enforced by the compiler and hardware with respect to the original source-level program—in contrast to hardware memory models, which only guarantee behaviors with respect to the compiled program.

An execution has a *data race* if two accesses are *conflicting* (they access the same shared, non-synchronization variable, and at least one is a write) and not ordered by the *happens-before relation*, a partial order that is the union of thread and synchronization order [20].

Generally speaking, memory models—including all of the memory models we discuss here—provide strong semantic guarantees for *data-race-free* (DRF) executions. In particular, DRF executions provide serializability of *synchronization-free regions* (SFRs) [2, 3, 23]. However, memory models differ in the guarantees they provide for *racy* (non-DRF) executions.

DRF0. The *data-race-free-0* (DRF0) memory model provides weak or undefined semantics for racy executions [3]. The Java and C++ memory models are both variants of DRF0 [2, 9, 25].

Despite much effort, languages and analyses fail to help eliminate all data races (e.g., [13, 17, 27]), so programs still have data races [2, 30]. Furthermore, programmers sometimes introduce data races intentionally for performance [32].

Java memory model. The Java memory model (JMM) defines the behavior of shared memory accesses on programs running in a Java virtual machine (JVM) [25]. Thus, the JMM applies not only to Java programs but also to programs written in other Java platform languages such as JRuby and Scala.

The JMM provides weak semantics for racy executions that preserve memory and type safety. The JMM introduces a concept called *causality* that must be respected by compiler transformations in order to avoid so-called "out-of-thin-air results" (a concept that is not well defined but refers to executions that experts generally agree should be avoided and can violate memory and type safety) [10, 25, 35]. Although the JMM guarantees memory and type safety, it still permits unintuitive results such as loads of "stale" and "future" stored values, i.e., threads can observe other threads' events happening in different orders.

Furthermore, researchers have shown that common JVM compiler optimizations can violate JMM causality rules and permit out-of-thin-air results [35]. Thus, JVMs do *not* actually enforce even the weak guarantees of the JMM, and no one knows how to guarantee causality (and thus memory and type safety) without severely limiting compiler optimizations [10, 35].

Sequential consistency. The *sequential consistency* (SC) memory model guarantees that threads' operations appear to interleave in a global order that respects program order [21]. While SC is stronger than the JMM, end-to-end SC does not provide a compelling reliability–performance tradeoff. SC is difficult for programmers to reason about, and it does not eliminate many unexpected behaviors. 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 ..." [2]. Despite being a fairly weak model, SC is not particularly cheap to provide: providing end-to-end SC requires restricting optimizations that would reorder memory accesses in both the compiler and hardware.

Statically bounded region serializability. Our prior work proposes a memory model based on *region serializability*: dynamically executed regions of code appear to execute together in program order, i.e., regions execute atomically. In our model, called *statically bounded region serializability* (SBRS), regions are synchronization free, intraprocedural, and acyclic [31]. That is, regions are bounded at synchronization operations, method calls, and loop back edges. SBRS offers a compelling reliability–performance tradeoff.

SBRS allows fewer behaviors than SC, simplifying the job of static and dynamic program analyses. Furthermore, SBRS enforces atomicity of regions that programmers may already expect to execute atomically; it eliminates real (atomicity) bugs that SC cannot eliminate [31]. Because regions are statically and dynamically bounded, SBRS is cost effective to enforce in software, as we describe next.

2.2 Enforcing SBRS

This section describes our prior work's approach for enforcing SBRS [31], which this paper refers to as *EnfoRSer-D* in order to distinguish its use of dynamic locks (locks associated with run-time objects) from the static locks (locks associated with static memory accesses) introduced in this paper. EnfoRSer-D modifies the JVM's dynamic compiler to transform statically bounded regions so that they execute atomically at run time. We note that these transformations are performed automatically during the JVM's just-in-time (JIT) compilation, so they are not visible to programmers, and programmers do not need to be aware of them.

EnfoRSer-D modifies the compiler to partition the program into statically bounded regions: regions bounded at synchronization operations, method calls, and loop back edges. To preserve SBRS, the compiler must not reorder memory accesses across region boundaries.

The general properties of EnfoRSer-D mentioned so far apply to all versions of EnfoRSer described in the paper. The rest of this section applies specifically to EnfoRSer-D.

EnfoRSer-D's dynamic locks and atomicity transformations. EnfoRSer-D's compiler transformation inserts instrumentation into regions to enforce *two-phase locking*: before each memory access, the compiler inserts instrumentation that acquires a *per-object lock* on the accessed object (or static field). To support per-object locks, EnfoRSer-D extends the JVM's object model so that each object has an extra header word and each static field has a class-level shadow word. A region does not release its acquired locks until the end of the region.

Two-phase locking guarantees atomicity, but it can deadlock if two regions attempt to acquire the same locks in different orders. EnfoRSer-D avoids deadlock—and reduces synchronization costs—by using *biased reader–writer locks* [12] for its per-object locks. These locks have the important property that a thread only releases locks when another thread wants to acquire the same lock. In other words, when a lock is not contended, a thread, T1, does not release the lock at the end of a region, avoiding the overhead of reacquiring the lock if the thread accesses the same object again. When another thread, T2, wants to acquire the lock (because it wants to access the object protected by the lock), thread T1 will release the lock at one of two times:

1. at the end of the region T1 is currently executing
2. if T1 attempts to acquire another lock held by any other thread

In the first situation, region atomicity is preserved, as the lock is held for the duration of the region. The second situation indicates a potential deadlock, so T1 "gives up" the lock to T2, allowing the latter to acquire the lock and breaking any locking cycle. Unfortunately, if this second situation arises, the principle of two-phase locking has been compromised—T1 did not hold its locks for the duration of the region—so region atomicity may have been violated.

To handle this potential violation of region atomicity, EnfoRSer-D's compiler transformation transforms regions so that they can safely restart whenever the executing thread gives up an acquired lock in the middle of a region. EnfoRSer-D uses one of two transformations. The *idempotent transformation* transforms regions so that they execute idempotently (i.e., without side effects) until they have successfully acquired all per-object locks. This transformation involves deferring the region's stores until the end of the region, which the transformation accomplishes by generating correct code using static program slicing in order to handle conditionally executed regions and potentially store–load aliasing. The *speculation transformation* transforms regions so they execute speculatively, backing up executed stores, so that the region can restore

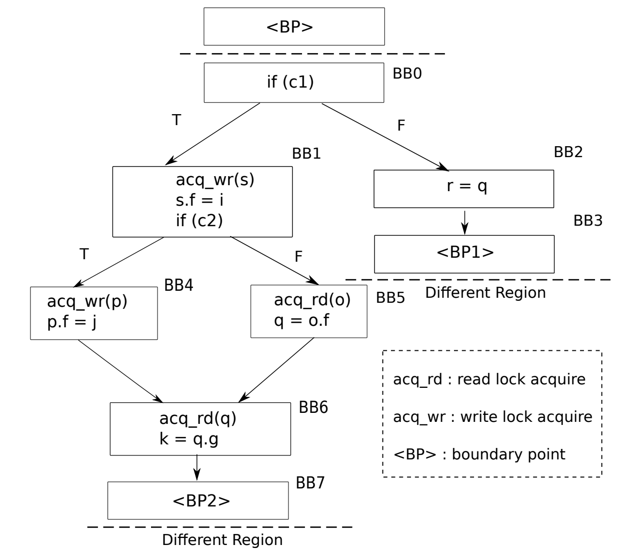

Figure 1. A statically bounded region instrumented by EnfoRSer-D with lock acquire operations, acq_rd() and acq_wr().

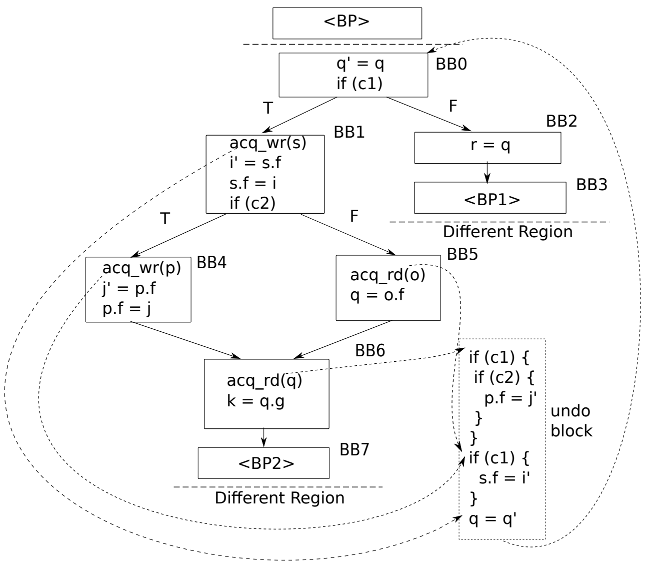

Figure 2. Region from Figure 1 after EnfoRSer-D performs its speculation transformation.

its initial state if it needs to restart. The speculation transformation involves significant complexity when generating restore code, relying on static program slicing to generate correct code. (We note that although these approaches resemble software transactional memory [19], EnfoRSer-D's focus on statically and dynamically bounded regions enables significantly better performance: statically bounded regions enable simpler instrumentation involving local variables instead of in-memory logs, and dynamically bounded regions permit EnfoRSer-D to forgo tracking read/write sets and instead restart conservatively on any potential conflict [31].)

Example. Figure 1 shows a statically bounded region after it has been instrumented with reader–writer lock acquires by EnfoRSer-D.[2] The points <BP>, <BP1>, and <BP2> each represent a

[2] Figures 1 and 2 are reused from our prior work [31].

region boundary point: a synchronization operation, method call, or loop back edge.

Figure 2 shows the region from Figure 1 after EnfoRSer-D's speculation transformation. We show the speculation transformation since it generally adds lower run-time overhead than the idempotent transformation [31], and this paper uses EnfoRSer-D exclusively with the speculation transformation. As the figure shows, the speculation transformation adds instrumentation before each memory store (`s.f = i` and `p.f = j`) that backs up the old value of the memory location in a new, dedicated local variable (`i'` and `j'`). The transformation also adds code at the beginning of the region to back up any local variables that are live into the region and are written by the region; in the figure, the region backs up `q` into a new local variable `q'`. The speculation transformation generates an "undo block" that restores the original values of stored-to memory locations and local variables. The transformation adds conditional jumps, shown as dotted lines in the figure, from each lock acquire to the appropriate point in the undo block. A lock acquire jumps to the undo block only if the acquire operation involves transferring ownership from another thread or threads, since that process permits any other thread to transfer ownership of another object's lock that the region may already have acquired.

EnfoRSer-D's run-time costs. EnfoRSer-D transforms regions to perform fine-grained locking at the object level. This approach avoids contention and achieves high scalability because locks only conflict when regions perform conflicting accesses to objects. However, EnfoRSer-D's approach has two main drawbacks that lead to high run-time overhead. First, EnfoRSer-D adds instrumentation costs to every program memory access (except for accesses that are statically redundant in a region or statically data race free; Section 4.1). Although biased reader–writer locks provide low overhead for the common case—non-conflicting lock acquires—this overhead is still significant when incurred at nearly every memory access [12]. Second, EnfoRSer-D's atomicity transformations (the idempotent and speculation transformations) add additional run-time costs in order to support retrying regions [31].

These two main run-time costs of EnfoRSer-D can be seen in Figure 2. First, each memory load and store is preceded by a lock acquire operation. Second, EnfoRSer-D has transformed the region to execute speculatively: the region backs up stored-to memory locations and locals, and it adds an undo block. Although by design the undo block executes infrequently, the conditional control flow edges add additional complexity that limits optimizations within the region. Furthermore, the additional local variables used to back up values are live into the undo block, which adds register pressure.

The rest of this paper explores an alternative to EnfoRSer-D's transformation that acquires *a single lock* for the entire region, avoiding both instrumentation at every memory access and transformations that ensure atomicity. However, acquiring a lock on the whole region—which involves acquiring the same lock for every pair of regions that potentially race with each other—adds high contention in many cases. The challenge is to apply these region locks judiciously.

3. Hybridizing Static and Dynamic Locks

This section describes the design of *EnfoRSer-H*, a version of EnfoRSer that enforces SBRS through a combination of per-object dynamic locks and per-region static locks. The high-level intuition of EnfoRSer-H is that using *static* locks to enforce SBRS incurs less instrumentation overhead than using dynamic locks (see the previous section for details of the overheads introduced by EnfoRSer-D), but using *dynamic* locks means that regions are less likely to conflict. Hence, EnfoRSer-H adopts a hybrid scheme that uses static locks to provide atomicity where we expect few conflicts (reducing the likelihood of expensive contention), while using dynamic locks when regions often execute concurrently (where static locks would result in significant contention). The following sections describe: (a) how SBRS can be enforced using per-site static locks, and how static locks be coarsened to the granularity of regions to reduce overhead; (b) how SBRS can be enforced using a combination of static and dynamic locks; and (c) a policy for selecting the right combination of static and dynamic locks.

3.1 EnfoRSer-S: Enforcing SBRS with Static Locks

We first present a version of EnfoRSer that enforces SBRS with static locks only, called EnfoRSer-S. EnfoRSer-S operates as follows: first, it instantiates a global set of static locks, L, for the program. Next, it associates *each access site* (hereafter called "site" for brevity) in a region with one of those static locks; the set of static locks acquired in a region r is $L(r)$. Because $L(r)$ is statically known, EnfoRSer-S can acquire all of the locks in the set at the beginning of the region, in some canonical order. Like in EnfoRSer-D, a region does not allow locks to be released in the middle of the region.

This strategy provides two-phase locking while avoiding the possibility of deadlock (because of the canonical ordering of static locks), without the need for complex atomicity transformations, as in EnfoRSer-D. However, this strategy alone does not suffice to guarantee atomicity. Consider two regions r_1 and r_2, with a site s_1 in r_1 that reads an object o, while a site s_2 in r_2 writes to that same object. To enforce SBRS, EnfoRSer-S must ensure that r_1 and r_2 cannot execute simultaneously. Note that EnfoRSer-D inherently provides this guarantee, due to its dynamic, per-object locking: because both s_1 and s_2 access o, they will both attempt to acquire a lock on o, triggering a conflict that EnfoRSer-D arbitrates. EnfoRSer-S, on the other hand, uses static, *per-site* locks; we must ensure that the locks that EnfoRSer-S acquires in r_1 and r_2 prevent the regions from executing simultaneously.

We note that a pair of sites $\langle s_1, s_2 \rangle$ presents a problem for enforcing SBRS only if three conditions are met: (i) both sites might access the same object; (ii) at least one of those accesses writes to the object; and (iii) the regions containing s_1 and s_2 could execute simultaneously in the original program. If any of those three conditions are not met, then executing r_1 and r_2 simultaneously does not violate atomicity. In other words, the problem arises if and only if s_1 and s_2 *race*. Hence, EnfoRSer-S can guarantee SBRS by ensuring that all pairs of sites that could race with each other *use the same static lock*. In other words, for any two regions r_a and r_b that contain sites s_a and s_b that race, EnfoRSer-S can guarantee SBRS by ensuring that $L(r_a) \cap L(r_b) \neq \emptyset$.

Hence, EnfoRSer-S uses the following strategy for *static lock assignment* when choosing which lock to associate with each site. It begins by using a sound static race detector to determine which sites might race with each other. EnfoRSer-S uses this information to construct an equivalence relation, $RACE$, over the set of sites in the program where s_1 $RACE$ s_2 if and only if:

1. s_1 races with s_2 according to the race detector (which is a symmetric property); or
2. $s_1 = s_2$; or
3. $\exists s_3 \,.\, s_1 \; RACE \; s_3 \wedge s_2 \; RACE \; s_3$

Note that s_a $RACE$ s_b does *not* imply that s_a and s_b race with each other according to the static race detection analysis. $RACE$ is an equivalence relation that adds reflexivity and transitivity to the symmetric property "(potentially) races with."

Then, for each equivalence class in $RACE$, EnfoRSer-S assigns a single static lock.[3] Hence, for each site s, EnfoRSer-S assigns it some lock l, which is associated with s's equivalence class in $RACE$, and every site s' that races with s has been assigned l as well. When this lock assignment algorithm is used with the locking strategy outlined above, EnfoRSer-S enforces SBRS using only static locks while avoiding deadlock.

Coarsening static locks. As described so far, the locking strategy for EnfoRSer-S requires that a lock be acquired for *every site* in a region. Acquiring a lock for every site in a region—including those that do not execute due to control flow—is likely to add overhead similar to EnfoRSer-D's, which also acquires a lock for every memory access. To mitigate this overhead, we observe that locks can be *coarsened*: while each equivalence class in $RACE$ must use a single lock to ensure correctness, there is no reason that different equivalence classes must be assigned different locks.

EnfoRSer-S thus uses a simple, region-based strategy for lock coarsening: in the pursuit of low overhead, it uses a *single* static lock for each region. In other words, all sites in a given region are assigned the same lock. To formalize this, let us define the $SRSL$ relation as an equivalence relation where s_1 $SRSL$ s_2 if and only if s_1 and s_2 are in the same region.[4] $SRSL \cup RACE$ is also an equivalence relation, and EnfoRSer-S ensures that each equivalence class in this combined relation uses the same lock.

Hence, each region r acquires a *single* lock when it begins, and any region r' that races with r (i.e., contains a site that races with a site in r) will acquire the same, single static lock when it begins.

3.2 EnfoRSer-H: Enforcing SBRS with Static and Dynamic Locks

While enforcing SBRS with EnfoRSer-S is sound, we note that it can be overly conservative in deciding whether to prevent two regions from executing simultaneously, and hence *over-serialize* execution. If two regions contain sites that *may* race with each other according to the static race detector, EnfoRSer-S will introduce locks that prevent those regions from executing concurrently. However, this locking is subject to three sources of imprecision:

1. Because of control flow in regions, locks may be acquired that prevent two regions from executing simultaneously even though during that particular execution of the regions, one or both of the racing sites may not actually execute.
2. Static race detectors identify sites that *may* race. At run time, these races may not occur because either (a) the inherent imprecision of static analysis means that these sites *never* race, or (b) even though under some circumstances the sites race, in this particular execution the two sites access different objects and so no race occurs.
3. Because EnfoRSer-S assigns locks according to the $RACE$ relation (more precisely, according to the $RACE \cup SRSL$ relation), two sites may use the same lock because they each race with a third site, even though these two sites can never race with each other.

Note that these sources of imprecision are not equally problematic: acquiring static locks when those locks are unlikely to result in conflicts does not cause much contention, while acquiring static locks that introduce unnecessary conflicts, results in over-serialization. Note, too, that none of these sources of imprecision afflict EnfoRSer-D: its per-object locks always acquire exactly the locks necessary to prevent two conflicting regions from executing simultaneously.

We thus introduce EnfoRSer-H, a *hybrid* version of EnfoRSer that uses both static locks at the region level and dynamic locks at the memory access level. For each equivalence class in $RACE \cup SRSL$, EnfoRSer-H uses either static or dynamic locks, depending on the tradeoff between instrumentation overhead (which favors static locks) and likelihood of conflict (which favors dynamic locks).

We note that the $SRSL$ ("same region (static locks)") relation applies only to accesses that use *static locks*. If two sites s_1 and s_2 are in the same region, s_1 $SRSL$ s_2 only if both s_1 and s_2 use static locks. $SRSL$ is thus defined in part according to EnfoRSer-H's choice of static versus dynamic locks for each site.

EnfoRSer-H proceeds directly from the definition of EnfoRSer-S. EnfoRSer-S uses a single, static lock to protect each equivalence class of $RACE \cup SRSL$. EnfoRSer-H allows some of those equivalence classes to be protected using EnfoRSer-D's dynamic, per-object locks instead. In other words, each site in the region is either protected by a static lock (which is acquired at the beginning of the region, a la EnfoRSer-S) or by a dynamic lock (which is acquired on demand, right before the access, a la EnfoRSer-D). As long as all the sites in a $RACE \cup SRSL$ equivalence class use the same *type* of lock (static or dynamic), atomicity of regions is still ensured.

Note that, if *any* site in a region uses dynamic locks, the site must be transformed using one of EnfoRSer-D's atomicity transformations to support restart. For regions that have a mix of static and dynamic locks, the static locks are acquired before the transformed region executes.

3.3 Choosing Which Locks to Use

Section 3.2 described how EnfoRSer-H can operate with different sites using different types of locks to enforce atomicity and hence provide SBRS. An obvious question to ask is how to determine which sites should use which type of lock. Static locks should be used when excessive serialization is unlikely—when sites do not often conflict. Dynamic locks, on the other hand, should be used when the benefit of determining more precisely *when* regions conflict makes up for higher instrumentation overheads. Because these are run-time properties, EnfoRSer-H uses profiling to collect information about the behavior of regions in the program. This profiling information is then used by a lock *assignment algorithm* that determines which type of lock should protect each site. This section describes that process.

Profiling data. Run-time profiling collects three pieces of data:

1. for each site s, the execution frequency of the region containing s;
2. for each site s, the number of conflicting lock acquires that occur when using a static lock, based on the $RACE$ equivalence class, in order to protect s's region; and
3. a mapping from sites to the region(s) that contain them statically (sites may appear in multiple regions statically, e.g., because of code expansion optimizations such as inlining).

The first two pieces of data are execution frequencies computed by compiler-inserted instrumentation. The third is computed by the just-in-time compiler.

In order to collect this data, each method is first compiled so that each site in a region is guarded by a static lock associated with its $RACE$ equivalence class—which is equivalent to EnfoRSer-S

[3] Note that some sites may not appear in RACE at all. These sites do not require locks, as they cannot lead to an SBRS violation.

[4] $SRSL$ is an acronym for "same region (static locks)." It applies only to sites that use static locks, not dynamic locks. This distinction is irrelevant for EnfoRSer-S, which only uses static locks, but is relevant for EnfoRSer-H (Section 3.2).

Algorithm 1 EnfoRSer-H's lock assignment algorithm.

```
1:  buildSRSLRelation()
2:  $initialCost \leftarrow$ estimateCost()
3:  for $Site\ s\ in\ RACE$ do
4:      $sType \leftarrow s$.getLockType()
5:      if $sType$ is static lock type then
6:          $s$.setLockType($DynamicLockType$)
            # Change the lock type to per-object type
7:          markAllSitesInRaceEqvClassAsDynamicLockType($s$)
8:          buildSRSLRelation()
9:          $currentCost \leftarrow$ estimateCost()
10:         if $currentCost \leq initialCost$ then
11:             $initialCost \leftarrow currentCost$
12:         else
13:             $s$.setLockType($StaticLockType$)
14:             resetAllSitesInRaceEqvClassAsStaticLockType($s$)
15:             buildSRSLRelation()
16:         end if
17:     end if
18: end for
```

without considering the $SRSL$ relation. The compiler inserts instrumentation at each static lock acquire to count execution frequency and the frequency of conflicting acquires. The compiler updates the mapping from sites to regions as it compiles each region.

Assigning locks. Under EnfoRSer-H, some sites are guarded by static locks while other sites use dynamic locks. The number of possible configurations for a hybrid implementation is 2^N, where N is the number of equivalence classes in the $RACE$ equivalence relation. Each equivalence class can be independently set to use static locks or dynamic locks, but all of the sites in an equivalence class must use the same type of lock for correctness. Note that in practice, there are fewer than 2^N possible configurations: for a given configuration, the use of the $SRSL$ relation to perform lock coarsening means that equivalence classes in $RACE$ that have sites that appear in the same region must use the same type of lock; assigning one $RACE$ class to use static locks may ultimately require that several other $RACE$ classes also use static locks.

EnfoRSer-H uses a greedy algorithm, as shown in Algorithm 1, to determine which $RACE$ classes should use static locks and which should use dynamic locks. The algorithm begins by assuming that all regions use static locks. Based on this assumption, it computes the associated $SRSL$ relation. Starting with this assignment, the algorithm computes an *estimated cost* according to a function estimateCost. This estimated cost is a function of the expected number of executed lock acquires and conflicting lock acquires. This section describes estimateCost in more detail below.

Given a lock configuration and an estimated instrumentation cost, the algorithm proceeds greedily. It iterates through each $RACE$ equivalence class, switching all of the class's sites to use dynamic locks (if they have not already been assigned dynamic locks). The algorithm propagates the effects of this switch by recalculating the $SRSL$ relation (since sites using dynamic locks are not included in $SRSL$). The algorithm then computes the cost of this new configuration. If the cost is less than the previous configuration, then the algorithm continues on to the next iteration with the new configuration, i.e., with the sites in the current $RACE$ equivalence class using dynamic locks. Otherwise, the algorithm continues with the previous configuration, i.e., with the sites in the current $RACE$ equivalence class using static locks.

While this greedy algorithm may not find the optimal (i.e., lowest estimated cost) configuration, our experience is that it is effective in finding better configurations when they exist. For example, our evaluation shows that in cases where profiling indicates that static locks incur many conflicts, the algorithm correctly chooses an assignment that instruments the program with mostly dynamic locks.

Estimating execution cost. At a high level, EnfoRSer-H predicts the costs that an execution will incur from acquiring locks, given profiling data and a candidate assignment of locks to be static or dynamic.

The basic intuition behind the estimate are two sources of overhead when enforcing SBRS using EnfoRSer-H:

1. The *fast-path cost*. This is the cost of acquiring the necessary locks to execute a region, assuming conflict freedom. For a region r, its estimated fast-path cost is:

   $$C_{fp}(r) = locks(r) \times T_{fp}$$

   where T_{fp} is the fast-path cost factor, and $locks(r)$ is estimated as follows.

   If the region r is protected by a static lock, one static lock acquire executes for the entire region. The function estimates this frequency as the *maximum* frequency of all sites in the region:

   $$locks(r) = \max_{s \in r}(\text{freq}(s))$$

   where freq(s) is number of times that a lock executed for site s according to profiling.

   If the region is protected by dynamic locks, the fast-path cost is estimated by summing the frequencies of all sites in the region:

   $$locks(r) = \sum_{s \in r}(\text{freq}(s))$$

   although it is an overestimate because it does not account for control flow.

2. The *conflict cost*. This is the cost of a conflict when a lock's ownership must be transferred to a new thread (Section 2.2). For a region r, its estimated conflict cost is:

   $$C_{confl}(r) = conflicts(r) \times T_{confl}$$

   where T_{confl} is the conflict cost factor, and $conflicts(r)$ is estimated as follows.

   If the region r is protected by a static lock, our experience suggests that merging locks using the $SRSL$ relation leads to a higher conflict rate than for the maximum conflicts of all sites in the region, so we instead compute the *sum* of all sites' conflicts in the region:

   $$conflicts(r) = \sum_{s \in r}(\text{conflicts}(s))$$

   where conflicts(s) is the number of conflicting lock acquires for s according to profiling.

   If r is protected by a dynamic lock, the algorithm cannot estimate the number of conflicts for dynamic locks because EnfoRSer-H collects profiling information using static locks based on $RACE$. Since dynamic locks tend to conflict infrequently compared to static locks, we assume no conflicts for dynamic locks:

   $$conflicts(r) = 0$$

Summing up all these costs across all regions provides an estimate of the instrumentation cost for a given lock assignment. Section 3.4 discusses the impact of the two cost factors, T_{fp} and T_{confl}.

Overview of EnfoRSer-H's optimization pipeline. Figure 3 illustrates EnfoRSer-H's optimization process. Each method is initially compiled so that each of its regions uses static locks based solely

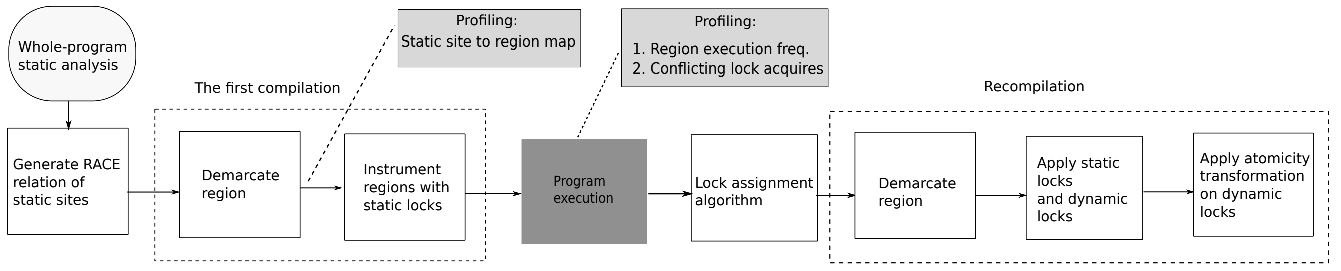
Whole-program static analysis
Profiling: Static site to region map
Profiling:
1. Region execution freq.
2. Conflicting lock acquires
The first compilation
Recompilation
Generate RACE relation of static sites
Demarcate region
Instrument regions with static locks
Program execution
Lock assignment algorithm
Demarcate region
Apply static locks and dynamic locks
Apply atomicity transformation on dynamic locks

Figure 3. EnfoRSer-H's optimization pipeline.

on the $RACE$ relation, as well as counters for collecting profile information. The second compilation uses the results of the assignment algorithm to use the chosen combination of static and dynamic locks.

3.4 Improving the Cost Estimate

This section describes a few extensions to the estimateCost function that help to improve its accuracy.

Conservative conflict prediction. We note that EnfoRSer-H's profiling counts a site's conflicts using static locks merged according to the $RACE$ equivalence class, but EnfoRSer-H chooses static locks so that all sites in the $RACE \cup SRSL$ equivalence class will use the same static lock. As a result, any two sites in this equivalence class that did not conflict with each other during profiling, may now conflict with each other with the coarsened locks.

To provide a better prediction of this effect, the estimateCost function computes the number of predicted conflicts for a site by taking the maximum number of conflicts measured for any site in the same $RACE \cup SRSL$ equivalence class:

$$computedConflicts(s) = \max_{s \in e} \text{profiledConflicts}(s)$$

where e is the $RACE \cup SRSL$ equivalence class for s.

Considering redundant lock acquires in the model. EnfoRSer-D and EnfoRSer-H use an optimization that removes redundant dynamic locks: if an earlier access in a region guarantees that the appropriate lock will already be held for a given access, the lock acquire for the latter access can be elided [31]. The cost model accounts for this effect by internally performing this *redundant lock elimination* optimization when computing the estimated fast-path cost for a region.

Ratio of costs of a conflict and a fast path. The two parameters in the cost model that drive the instrumentation overhead estimates are the cost of a fast path, T_{fp}, and the cost of a conflict on statically-locked regions, T_{confl}. The ratio of these two parameters captures how biased the assignment algorithm is toward static versus dynamic locks: static locks incur conflicts, and are selected against if T_{confl} is high, while dynamic locks require more fast paths, and are selected against if T_{fp} is high. Adjusting the ratio between these parameters leads to different lock assignment outcomes. In our experiments, we have found that EnfoRSer-H's performance is quite stable across a wide range for the ratio of T_{confl} to T_{fp} (50–300), suggesting that it is most important to identify a key set of outlier regions when performing lock assignment.

4. Evaluation

This section first describes our implementation and experimental methodology. It then compares the run-time characteristics and performance of EnfoRSer-D, EnfoRSer-S, and EnfoRSer-H.

4.1 Implementation

We have implemented the EnfoRSer configurations in Jikes RVM 3.1.3 [5, 6],[5] a Java virtual machine that provides performance competitive with commercial JVMs [7]. Our implementation builds on the publicly available implementation of our prior work, which provides only the EnfoRSer-D algorithm [31]. We have now made our implementations of EnfoRSer-D, EnfoRSer-S, and EnfoRSer-H publicly available on the Jikes RVM Research Archive.[6]

Jikes RVM uses two just-in-time compilers at run time. The first time a method executes, Jikes RVM compiles it with the *baseline* compiler, which generates unoptimized machine code directly from Java bytecode. When a method becomes hot, Jikes RVM compiles it with the *optimizing* compiler at successively higher optimization levels. As in our prior work, we modify *only the optimizing compiler* to enforce SBRS, since EnfoRSer's transformations require a compiler internal representation [31]. While this approach is technically unsound, it approximates the performance of a fully sound implementation because, by design, execution spends most of its time in optimized code.

As in our prior work, the compiler limits reordering across region boundaries (synchronization operations, method calls, and loop back edges) in order to preserve SBRS. After letting the compiler perform some standard optimizations, the optimizing compiler performs EnfoRSer transformations, followed by additional standard optimizations, which help to "clean up" the code generated by EnfoRSer's transformations [31].

4.2 Methodology

Profiling and recompilation. EnfoRSer-S and EnfoRSer-H require profile information. Both EnfoRSer-S and EnfoRSer-H require information from the optimizing compiler about the $SRSL$ relation, i.e., which sites are compiled into which regions. Although in theory $SRSL$ could be computed statically, in practice inlining decisions and other optimizations expand regions and increase sites which appear in the same statically bounded regions. In addition, EnfoRSer-H relies on run-time profiling to compute the number of lock acquires and conflicts when using static locks. Both EnfoRSer-S and EnfoRSer-H initially choose static locks based only on the $RACE$ relation, which enables collecting run-time profile information about conflicts within each $RACE$ equivalence class.

In theory, an implementation could perform online profiling, updating $SRSL$ and adjusting the locking strategy for sites on the fly. This approach would require recompiling all methods affected by the use of new locks and new locking strategies. While such an approach is possible, we have not undertaken this engineering effort. Instead, our implementation runs two iterations of each program. The first iteration collects profile information: it executes

[5] `http://www.jikesrvm.org`

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

	EnfoRSer-D		EnfoRSer-S		EnfoRSer-H					
	Total		Total		Static locks		Dynamic locks		Total	
	Executed	Confl.	Executed	Confl.	Executed	Confl.	Executed	Confl.	Executed	Confl.
hsqldb6	4.7×10^8	5.9×10^5	2.1×10^8	5.2×10^5	2.3×10^7	3.8×10^3	4.4×10^8	5.6×10^5	4.7×10^8	5.7×10^5
lusearch6	1.5×10^9	3.9×10^3	3.8×10^8	3.3×10^8	4.1×10^7	3.7×10^2	1.4×10^9	3.6×10^3	1.4×10^9	4.0×10^3
xalan6	6.1×10^9	1.4×10^7	2.3×10^9	1.5×10^9	2.0×10^8	1.7×10^2	6.1×10^9	1.5×10^7	6.3×10^9	1.5×10^7
avrora9	4.0×10^9	4.0×10^6	1.1×10^9	7.9×10^8	3.8×10^8	1.0×10^1	3.9×10^9	4.0×10^6	4.3×10^9	4.0×10^6
luindex9	1.3×10^8	7.1×10^1	6.8×10^7	1.8×10^1	6.0×10^7	3.0×10^0	6.5×10^6	3.3×10^1	6.6×10^7	3.6×10^1
lusearch9	1.5×10^9	1.2×10^4	4.7×10^8	4.4×10^8	1.0×10^8	3.1×10^3	1.4×10^9	9.7×10^2	1.5×10^9	4.0×10^3
sunflow9	7.2×10^9	1.6×10^4	3.3×10^9	6.0×10^7	2.1×10^9	3.2×10^2	2.3×10^9	1.2×10^4	4.4×10^9	1.2×10^4
xalan9	4.2×10^9	2.0×10^7	1.3×10^9	6.2×10^8	1.2×10^8	3.4×10^4	4.1×10^9	2.0×10^7	4.2×10^9	2.0×10^7
pjbb2000	1.2×10^9	9.5×10^5	6.2×10^8	2.6×10^8	1.2×10^8	1.4×10^3	1.1×10^9	9.5×10^5	1.2×10^9	9.5×10^5
pjbb2005	6.3×10^9	4.0×10^8	1.8×10^9	1.0×10^9	1.1×10^9	2.5×10^3	4.4×10^9	4.2×10^8	5.5×10^9	4.2×10^8

Table 1. Dynamic lock acquire operations executed, and the number of them that result in a conflicting lock state transition requiring coordination among threads. For EnfoRSer-H, the table shows the breakdown for static and dynamic locks.

code compiled entirely with static locks based only on the $RACE$ equivalence class. Note that the $SRSL$ equivalence classes are not known during the first iteration, so the first iteration cannot use code compiled based on $SRSL$.

After the first iteration completes, Jikes RVM recompiles all methods that have been compiled by the optimizing compiler; when the optimizing compiler recompiles a method during this phase, it uses the profile information from the first iteration. For the EnfoRSer-S configuration, the compiler uses the $SRSL$ relation in order to compute static region locks based on $RACE \cup SRSL$. EnfoRSer-H runs the assignment algorithm, which uses run-time profile information about static locks in order to decide whether to use static or dynamic locks for each site. For sites that use static locks, EnfoRSer-H uses the $RACE \cup SRSL$ relation in order to use one lock for each region while preserving correctness. EnfoRSer-D ignores the profiling information and uses dynamic per-object locks, and is thus equivalent to our prior work [31].

The second iteration then executes using the recompiled methods. Our evaluation reports the cost of the second iteration only.

This two-iteration methodology thus represents an optimistic performance measurement: it excludes compilation time and the cost of the assignment algorithm, and it uses profile information on a prior identical run. On the other hand, this methodology does not account for the fact that EnfoRSer-S and EnfoRSer-H have the potential to *reduce* compilation time (and thus execution time, since JIT compilation can affect program execution time) by avoiding EnfoRSer-D's complex atomicity transformations.

Static race detection. EnfoRSer-S and EnfoRSer-H rely on conservative static data race detection in order to compute the $RACE$ relation. We use Naik et al.'s whole-program static data race detector, implemented as the publicly available tool *Chord*.[7] We disable Chord's lockset analysis, which is unsound (has false negatives) because it uses may-alias analysis [28].[8] The resulting analysis identifies accesses as data race free (DRF) based on static thread escape analysis and fork–join analysis [28]. We use Chord's built-in mechanism to handle reflection, which resolves reflective calls by running the input program prior to the static analysis.

In prior work, we used the same configuration of Chord in order to identify definitely DRF accesses, which can forgo EnfoRSer-D locks [31]. For a fair comparison, our evaluation continues to perform this optimization for EnfoRSer-D.

Benchmarks. We evaluate the EnfoRSer implementations using benchmarked versions of large, real-world multithreaded applications: the DaCapo benchmarks [8] versions 2006-10-MR2 and 9.12-bach (2009), distinguished with names suffixed with '6' and '9', using the large workload size; and fixed-workload versions of SPECjbb2000 and SPECjbb2005.[9] We exclude benchmarks that unmodified Jikes RVM cannot execute. In addition, we exclude eclipse6 since Chord fails when analyzing it. We exclude jython9 and pmd9 since Chord does not report any potential data races in these programs; we suspect that jython9, which has limited multithreaded behavior, in fact has no data races, while Chord does not understand pmd9's multithreading based on futures. We cross-checked Chord's results with a dynamic race detector's output [11, 17]; we found one class (in jbb2005) that Chord does not analyze at all (for unknown reasons), so EnfoRSer-S, EnfoRSer-D and EnfoRSer-H fully instrument this class.

Platform. The experiments execute a high-performance configuration of Jikes RVM with the default high-performance garbage collector (FastAdaptiveGenImmix). The experiments execute on an Intel Xeon E5-4620 system with four 8-core processors (32 cores total) running Linux 2.6.32.

4.3 Run-Time Characteristics

Table 1 reports how many lock acquires execute, and how many of them incur a conflicting transition, for each EnfoRSer configuration. Each result is the mean of five trials of a statistics-gathering configuration of EnfoRSer-D, EnfoRSer-S, or EnfoRSer-H. For each configuration, the first column under *Total* reports the total number of lock acquire operations executed, and the second column reports how many of those operations result in a conflicting lock state transition, requiring coordination among threads.

As expected, EnfoRSer-S acquires a single lock per region, rather than per memory access, so it performs fewer lock acquires than EnfoRSer-D. However, EnfoRSer-S incurs many more conflicts than EnfoRSer-D, except for two programs (hsqldb6 and luindex9). EnfoRSer-S incurs many more conflicts than EnfoRSer-D because of the imprecision of detecting conflicts wtih static locks, for reasons described in Section 3.2.

For luindex9 and sunflow9, EnfoRSer-H achieves its desired goal, reducing lock acquires by 39–49% relative to EnfoRSer-D, without substantially affecting the fraction of conflicts, which is already very low. As Section 4.4 shows, EnfoRSer-H is able to lower run-time overhead relative to EnfoRSer-D for these programs.

[7] `http://pag.gatech.edu/chord`

[8] We have confirmed with Naik that the sound version of Chord's analysis, which uses conditional must-not alias analysis, is not available [27].

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

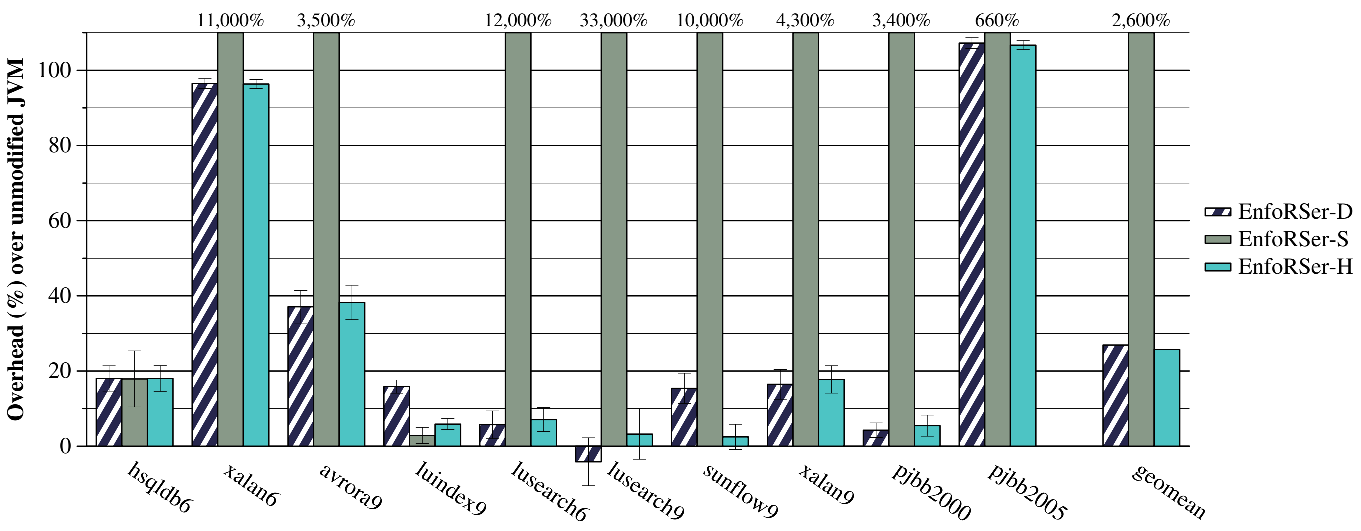

Figure 4. Run-time overhead of providing SBRS with EnfoRSer configurations that use dynamic locks, static locks, or a hybrid of both.

For other programs, EnfoRSer-H does not substantially affect how many lock acquire operations execute compared with EnfoRSer-D. At the same time, EnfoRSer-H does not significantly increase total lock acquires nor conflicting lock acquires. EnfoRSer-H does increase the total number of lock acquires slightly for xalan6 and avrora9 compared with EnfoRSer-D; this possibly unintuitive result is a side effect of the fact that EnfoRSer-H's static lock acquires execute at the beginning of a region (i.e., they are not sensitive to control flow in the region), while EnfoRSer-D's dynamic lock acquires execute only when their corresponding access executes.

We note that for pjbb2005, EnfoRSer-H reduces lock acquire operations compared with EnfoRSer-D. However, EnfoRSer-H also incurs more conflicts than EnfoRSer-D; the rate of conflicts is high for pjbb2005, so even a modest increase can incur high overhead, which may explain why our performance evaluation shows no statistically significant difference in overhead between the two configurations for pjbb2005 (Section 4.4).

4.4 Performance

Figure 4 shows the overhead of EnfoRSer-D, EnfoRSer-S, and EnfoRSer-H over baseline execution. Each bar is the mean of at least 10 trials; each bar has a 95% confidence interval centered at the mean.[10] As in Section 4.3, for each configuration including the baseline (unmodified Jikes RVM), Jikes RVM executes two iterations. The first iteration always uses static locks based solely on the $RACE$ relation. Before the second iteration, the optimizing compiler recompiles methods according to the EnfoRSer configuration: EnfoRSer-S and EnfoRSer-H compute $RACE \cup SRSL$ equivalence classes, and EnfoRSer-H uses the assignment algorithm to choose a combination of static and dynamic locks. In the baseline configuration, the optimizing compiler recompiles all methods without any instrumentation.

For most programs, EnfoRSer-S incurs substantially higher overhead than EnfoRSer-D, which is a result of EnfoRSer-S causing many more conflicting lock acquires than EnfoRSer-D (Table 1). EnfoRSer-S incurs an average overhead of 2600% overhead (a 27X slowdown) over baseline execution. This result shows that static locks must be applied judiciously in order to be effective.

[10] For some programs with high execution time variance, we have run additional trials in an effort to reduce confidence interval sizes.

EnfoRSer-S outperforms or matches EnfoRSer-D's performance for hsqldb6 and luindex9, which is to be expected from the low number of conflicts it occurs in Table 1.

Overall, EnfoRSer-H is able to improve the performance of enforcing SBRS compared with EnfoRSer-D, but the average improvement is modest: from 27% to 26%, a 4% reduction in overhead. This result is unsurprising given the results from Table 1: the reduction is modest overall, and significant benefits from EnfoRSer-H are limited to a few programs.

For two programs, luindex9 and sunflow9, EnfoRSer-H reduces overhead substantially compared with EnfoRSer-D, 63% and 87% reduction in overhead, respectively, over the baseline. Note that for luindex9, EnfoRSer-S also incurs low overhead compared to EnfoRSer-D since it reduces lock acquires without increasing the number of conflicting acquires, as this program inherently has few conflicts. This result follows directly from Table 1, which shows that EnfoRSer-H performs about half as many lock acquire operations as EnfoRSer-D for each of these programs.

For all other programs, EnfoRSer-H does not perform significantly worse than EnfoRSer-D. (EnfoRSer-H appears to perform slightly worse for xalan6, lusearch9, and xalan9, but the confidence intervals overlap.) This outcome results from EnfoRSer-H's assignment algorithm finding relatively few executing regions that can use static region locks without incurring significant contention (as Table 1 suggests), so it conservatively uses dynamic per-object locks in most cases, and thus EnfoRSer-H's performance resembles EnfoRSer-D's performance in these cases.

These results show the opportunities for and limitations of hybridizing locks to enforce SBRS efficiently. While static locks reduce the number of executed lock acquires compared with dynamic locks, their conservatism introduces many false conflicts. EnfoRSer-H judiciously uses static locks only where profiling and the cost model predict few enough conflicts to not outweigh the reduction in lock acquires, achieving better performance than either EnfoRSer-D or EnfoRSer-S can achieve alone. While the overall performance improvement is modest, EnfoRSer-H generally does not hurt performance relative to EnfoRSer-D or EnfoRSer-S, and it has the potential to improve performance significantly in cases that can benefit from a hybrid of static and dynamic locks.

5. Related Work

This section covers related work *other than* memory models and our prior work on EnfoRSer (called EnfoRSer-D in this paper), which Section 2 covered.

Static locking. To provide deterministic replay for racy programs, *Chimera* records an execution by recording not only its synchronization operations but also the ordering between accesses involved in data races [22]. Chimera uses the same static lock for each pair of accesses that potentially race with each other, according to conservative static analysis, similar to EnfoRSer-S's selection of static locks based on the $RACE$ relation. Using static locks slows programs by more than an order of magnitude. To reduce the overhead of acquiring a lock at each potentially racy access, Chimera coarsens lock granularity in cases where profiling predicts that two regions using the same lock will contend infrequently. While lock coarsening is similar in spirit to EnfoRSer-S and EnfoRSer-H's use of the $SRSL$ relation to acquire a single lock for each region, Chimera's coarsening can potentially lead to high contention because its regions are in general neither statically nor dynamically bounded. Chimera relies on symbolic execution in order to associate address ranges with coarsened locks to minimize false conflicts. This approach is mostly beneficial to avoid repeated false conflicts inside loops. EnfoRSer-H avoids such optimizations since loop back edges are natural boundaries of statically bounded regions.

Aside from the fact that Chimera focuses on recording cross-thread dependencies, whereas our work enforces atomicity, the two approaches differ in two important ways. First, Chimera uses only static locks associated with static code locations (sites or regions); it makes no use of what we call "dynamic locks": locks associated with shared memory (objects). Although Chimera can achieve reasonable performance using static locks, our results show that EnfoRSer-S experiences very high contention with static locks alone, suggesting that Chimera's evaluated programs have significantly less contention than our evaluated programs. Second, since Chimera uses only static locks, it does not investigate hybridizing static and dynamic locks, which is the main contribution of our work.

Utilizing static analysis. Much prior work has used static analysis to identify definitely data-race-free (DRF) or thread-local accesses, in order to reduce instrumentation costs [15, 16, 22, 31, 34]. Beyond distinguishing definitely DRF and potentially racy accesses, both this work and Chimera utilize information about potentially racy *pairs* of accesses in order to select a static lock to guard both accesses [22].

Prior work has used static analysis to identify accesses where instrumentation would be "redundant" due to preceding instrumentation for the same object [12, 18]. Our prior EnfoRSer work also identifies and eliminates instrumentation that would be redundant due to earlier instrumentation in the same region, using intraprocedural dataflow analysis [31]. In this paper, the compiler makes use of redundancy analysis for dynamic per-object locks.

Hybridizing locking mechanisms. This work targets a major source of overhead of enforcing SBRS: the *overhead* incurred by EnfoRSer-D to perform a lock acquire operation at every potentially racy memory access. An orthogonal cost is the performance penalty incurred by biased reader–writer locks for conflicting lock acquires [12, 31]. In other work, we have targeted this separate problem; that work chooses between *biased* and *unbiased* reader–writer locks based on run-time profile information [14]. EnfoRSer-S and EnfoRSer-H could make use of that complementary approach in order to use a combination of biased and unbiased reader–writer locks for both dynamic and static locks.

Other prior work has combined synchronization mechanisms adaptively. For example, Usui et al. combine lock-based mutual exclusion and software transactional memory (STM) [33]. Abadi et al. present an STM that adaptively changes how it detects conflicts for non-transactional accesses, depending on whether transactions access the same objects as non-transactional code [1].

Region serializability. Researchers have either enforced atomicity of code regions [4, 29] or checked violations of atomicity of code regions [23, 24, 26]. Prior work explores checking violations of serializability of full or bounded synchronization-free regions (SFRs) but requires custom hardware [23, 26]. Enforcing full SFR serializability in software is possible using replication, but it adds prohibitive overhead without additional cores [29]. *Atom-Aid* and *BulkCompiler* eliminate atomicity violations and/or provide sequential consistency by executing chunks of instructions atomically, but rely on hardware extensions [4, 24].

6. Conclusion

Statically bounded region serializability (SBRS) is a memory model that provides measurably stronger guarantees than the Java memory model. But prior work's EnfoRSer-D approach requires acquiring a lock at every memory access, which adds significant run-time overhead even for regions that are not generally involved in conflicts. This paper introduces EnfoRSer-H, which judiciously chooses a combination of static locks that provide lower instrumentation overhead but more conflicts than EnfoRSer-D's dynamic locks. While EnfoRSer-H's average performance benefit over EnfoRSer-D is small, EnfoRSer-H provides significant improvement in cases that it can benefit, without negatively impacting applications, demonstrating the potential for hybrid synchronization to provide strong end-to-end memory models for the Java platform.

Acknowledgments

We thank the anonymous reviewers for valuable feedback on the text. We thank Swarnendu Biswas, Jipeng Huang, Yan Wang, and Minjia Zhang for helpful discussions.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *PPoPP*, pages 185–196, 2009.

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

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

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

[6] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[7] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, 2015. To appear.

[8] 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.

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

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

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

[12] 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.

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

[14] 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.

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

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

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

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

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

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

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

[23] 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.

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

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

[26] 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.

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

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

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

[30] K. Poulsen. SecurityFocus News: Tracking the blackout bug, 2004.

[31] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, pages 561–575, 2015.

[32] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic Recognition of Synchronization Operations for Improved Data Race Detection. In *ISSTA*, pages 143–154, 2008.

[33] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *PACT*, pages 3–14, 2009.

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

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