## Title

DRF x : An Understandable, High Performance, and Flexible Memory Model for Concurrent Languages



## Journal

ACM Transactions on Programming Languages and Systems, 38(4)

## ISSN

0164-0925

## Authors

Marino, Daniel
Singh, Abhayendra
Millstein, Todd
et al.

## Publication Date

2016-10-13

## DOI

10.1145/2925988

Peer reviewed

# DRF*x*: An Understandable, High Performance, and Flexible Memory Model for Concurrent Languages

DANIEL MARINO, Symantec Research Labs
ABHAYENDRA SINGH, Google Inc.
TODD MILLSTEIN, University of California, Los Angeles
MADANLAL MUSUVATHI, Microsoft Research, Redmond
SATISH NARAYANASAMY, University of Michigan, Ann Arbor

The most intuitive memory model for shared-memory multi-threaded programming is *sequential consistency* (SC), but it disallows the use of many compiler and hardware optimizations and thus affects performance. Data-race-free (DRF) models, such as the C++11 memory model, guarantee SC execution for data-race-free programs. But these models provide no guarantee at all for racy programs, compromising the safety and debuggability of such programs. To address the safety issue, the Java memory model, which is also based on the DRF model, provides a weak semantics for racy executions. However, this semantics is subtle and complex, making it difficult for programmers to reason about their programs and for compiler writers to ensure the correctness of compiler optimizations.

We present the DRF*x* memory model, which is simple for programmers to understand and use while still supporting many common optimizations. We introduce a *memory model (MM) exception* that can be signaled to halt execution. If a program executes without throwing this exception, then DRF*x* guarantees that the execution is SC. If a program throws an MM exception during an execution, then DRF*x* guarantees that the program has a data race. We observe that SC violations can be detected in hardware through a lightweight form of conflict detection. Furthermore, our model safely allows aggressive compiler and hardware optimizations within compiler-designated program regions. We formalize our memory model, prove several properties of this model, describe a compiler and hardware design suitable for DRF*x*, and evaluate the performance overhead due to our compiler and hardware requirements.

CCS Concepts: • **Computer systems organization → Multicore architectures**; • **Software and its engineering → Parallel programming languages**; **Concurrent programming languages;**

Additional Key Words and Phrases: Sequential consistency, memory models, DRFx

**ACM Reference Format:**
Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2016. DRF*x*: An understandable, high performance, and flexible memory model for concurrent languages. ACM Trans. Program. Lang. Syst. 38, 4, Article 16 (September 2016), 40 pages.
DOI: http://dx.doi.org/10.1145/2925988

This work is supported by the National Science Foundation under awards CNS-0725354, CNS-0905149, and CCF-0916770 as well as by the Defense Advanced Research Projects Agency under award HR0011-09-1-0037. This work was performed while the first author was at the University of California, Los Angeles. Portions of this work were published previously in Marino et al. [2010] and Singh et al. [2011a]. This journal article synthesizes the material published earlier [Marino et al. 2010; Singh et al. 2011a].
Authors' addresses: D. Marino (current address), Symantec Corporation, 900 Corporate Pointe, Culver City, CA 90230; A. Singh (current address), Google Inc., 1600 Amphitheatre Parkway Mountain View, CA 94043; S. Narayanasamy, Computer Science Department, University of Michigan, Ann Arbor; T. Millstein, Computer Science Department, University of California, Los Angeles; M. Musuvathi, Microsoft Research, Redmond.

DOI: http://dx.doi.org/10.1145/2925988

## 1. INTRODUCTION

A memory consistency model (or simply *memory model*) determines the order in which memory operations of a program appear to execute and, thus, the possible values a read can return. Memory models impose a performance vs. programmability tradeoff. Weak memory models allow more compiler and hardware optimizations and thus improve runtime efficiency, while strong memory models simplify program behavior and hence make it easier to build robust software.

A natural memory model for shared-memory concurrency is *sequential consistency* (SC) [Lamport 1979]. The SC memory model, unlike the weaker memory models that are prevalent today in both hardware and programming languages, enforces two crucial programming abstractions: sequential composition and shared memory. Sequential composition allows programmers to assume that the computer executes instructions in the order specified in the program, despite the intricacies of compiler and hardware optimizations. The shared memory abstraction allows programmers to assume that there is a single logical copy of a variable that can be atomically read and written, despite the complexity of the memory hierarchy. Almost all programmers assume SC semantics when writing and reasoning about programs. Even expert programmers typically design their programs to be correct under SC and then insert synchronization primitives (such as fences) to prevent non-SC behaviors.

Given the intuitiveness of SC and the difficulty of reasoning about weak memory models, it should come as a surprise that none of today's concurrent programming languages support the SC memory model. Instead, they support variations of a weaker model called Data-race-free-0 (DRF0), which only guarantees SC for programs that are *data-race free*, with no guarantees for other programs. The data-race freedom requirement allows the compiler and hardware to perform a wide range of sequentially valid (i.e., correct on a single thread in isolation) optimizations without violating SC. Unfortunately, guaranteeing data-race freedom requires a significant amount of discipline on behalf of the programmer: *Every* memory location that can be accessed simultaneously by multiple threads (where at least one access is a write) must be explicitly annotated as *synchronization*. A single oversight—forgetting to acquire a lock, acquiring the wrong lock, or simply forgetting to annotate a variable as synchronization—can introduce non-SC program behaviors that are hard to understand and debug.

Our position in this article is that given the already-difficult task of reasoning about concurrent programs, programmers should *never* be required to reason about behaviors other than SC, even if this comes at some performance cost. To that end, in this article we explore a new memory model called DRF*x* that is a middle ground between SC and DRF0. Specifically, any language supporting SC also supports DRF*x* (but not vice versa in general), and any language supporting DRF*x* also supports DRF0 (but not vice versa in general). DRF*x* allows most DRF0 optimizations but only within bounded regions of code. Unlike DRF0, DRF*x* limits the effects of these optimizations on program behavior by halting potentially non-SC executions with an exception. This fail-stop semantics is the *only* non-SC behavior exposed to the programmer. Moreover, an exception indicates the presence of a data race in the program, which can be fixed to prevent the exception in the future.

One way to view DRF*x* is as an "optimistic" variant of SC, allowing most DRF0-compliant optimizations but then halting the program if this leads to non-SC behavior. An alternative approach that we have explored in follow-on research [Marino et al. 2011; Singh et al. 2012] is a "pessimistic" style, which modifies the compiler and hardware to ensure SC for all program executions. Specifically, we show that by carefully restricting compiler and hardware optimizations on non-thread-local accesses and by relaxing the ISA to allow the compiler to inform the hardware about thread-local accesses, SC is achievable with acceptable performance overheads.

We believe that both approaches to providing SC reasoning for programmers are valuable and worth further study, since there are a complex set of tradeoffs between them. For example, SC provides a simpler programming model than DRF*x* since the former ensures SC behavior without raising exceptions. On the other hand, as we will see, DRF*x* in fact ensures a stronger property than SC for exception-free executions, a form of *region serializability*, which can potentially simplify reasoning about thread interactions. As another example, DRF*x* allows most DRF0 optimizations and can easily incorporate future advances in DRF0-compliant optimizations but only within the bounded scope of a single region of code. On the other hand, an SC compiler and hardware can support arbitrary optimizations on thread-local variables, without having to respect any region boundaries. Given these complex tradeoffs, which we expand on in Section 8, we believe that both SC and DRF*x* are viable options as a strong memory model for multithreaded programming. Either one would be a significant improvement over DRF0.

### 1.1. The DRF*x* Memory Model

DRF*x* assumes (like DRF0) that data races are errors, and it dynamically detects and halts executions that exhibit a data race [Adve et al. 1991; Elmas et al. 2007; Boehm 2009]. Such detection ensures that a program execution is SC unless it is halted by the data-race detector. There are two major challenges in this approach. First, the data-race detection must be precise, neither allowing a program to complete its execution after a data race nor allowing a data-race-free execution to be erroneously rejected. Precise data-race detection in software is very expensive, even with recent optimizations [Flanagan and Freund 2009], and hardware solutions [Adve et al. 1991; Muzahid et al. 2009] are quite complex. Second, it is not sufficient to detect data races in the *compiled* program; rather, giving an SC guarantee to programmers requires that data races be detected relative to the original *source* program.

Our DRF*x* memory model addresses both of the challenges. We leverage the observation of Gharachorloo and Gibbons [1991] that to provide a useful guarantee to programmers, it suffices to detect only the data races that cause SC violations. They illustrate that such detection for a compiled program can be much simpler than full-fledged data-race detection. We describe a compiler and hardware co-design that lifts their guarantees from the binary to the source program while retaining high performance and implementation flexibility.

We introduce the notion of a dynamic *memory model (MM) exception* which halts a program's execution. DRF*x* guarantees two key properties for any program P:

—**Data-Race Completeness**: If an execution is terminated with an MM exception, then P has a data race.
—**SC Soundness**: If an execution is not terminated with an MM exception, then that execution is SC.

Together these two properties imply the DRF0 property: data-race-free programs obtain SC semantics (and are never terminated with an MM exception). However, unlike DRF0, the SC Soundness property allows programmers to safely reason about *all* programs, whether data-race free or not, using SC semantics. Finally, the Data-Race Completeness property ensures that MM exceptions cannot be raised spuriously but only when the program has a data race.

While these properties provide strong guarantees to programmers, they are carefully designed to admit implementation flexibility. For example, DRF*x* allows an MM exception to be thrown even if SC is not violated, as long as the original program has a data race. This is an acceptable result since, as with the DRF0 memory model, we consider a data race to be a programmer error. Conversely, DRF*x* also allows a data-racy execution

to continue without exception as long as it does not violate SC. As we will see, our compiler and hardware designs make good use of this flexibility.

SC Soundness requires only that an SC violation will cause execution to halt with an MM exception *eventually*, which also provides implementers significant flexibility. However, an execution's behavior is undefined between the point at which the SC violation occurs and the exception is raised. The DRF*x* model therefore guarantees an additional property:

—**Safety:** If an execution of P invokes a system call, then the observable program state at that point is reachable through an SC execution of P.

Intuitively, the above property ensures that any system call in an execution of P would also be invoked with exactly the same arguments in some SC execution of P. This property ensures an important measure of safety and security for programs by prohibiting undefined behavior from being externally visible.

### 1.2. A Compiler and Hardware Design for DRF*x*

Gharachorloo and Gibbons [1991] describe a hardware mechanism to detect SC violations. Their approach dynamically detects *conflicts* between concurrently executing instructions. Two memory operations are said to conflict if they access the same memory location, at least one operation is a write, and at least one of the operations is not a synchronization access. While simple and efficient, their approach guarantees the SC Soundness and Race Completeness properties with respect to the *compiled* version of a program but does not provide any guarantees with respect to the original *source* program [Gharachorloo and Gibbons 1991; Ceze et al. 2009].

A key contribution of DRF*x* is the design and implementation of a detection mechanism for SC violations that properly takes into account the effect of both compiler optimizations and hardware reorderings while remaining lightweight and efficient. The approach employs a novel form of cooperation between the compiler and the hardware. We introduce the notion of a *region*, which is a single-entry, multiple-exit portion of a program. The compiler partitions a program into regions, and both the compiler and the hardware may only optimize within a region. Each synchronization access must be placed in its own region, thereby preventing reorderings across such accesses. It is also required that each system call be placed in its own region, which allows DRF*x* to guarantee the Safety property. Otherwise, a compiler may choose regions in any manner in order to aid optimization and/or simplify runtime conflict detection. Within a region, both the compiler and hardware can perform most standard sequentially valid optimizations. For example, unrelated memory operations can be freely reordered within a region, unlike the case for the traditional SC model.

To ensure the DRF*x* model's SC Soundness and Race Completeness properties with respect to the original program, it suffices to detect *region conflicts* between concurrently executing regions. Two regions $R_1$ and $R_2$ conflict if there exists a pair of conflicting operations $(o_1, o_2)$ such that $o_1 \in R_1$ and $o_2 \in R_2$. Such conflicts can be detected using runtime support similar to conflict detection in transactional memory (TM) systems [Herlihy and Moss 1993]. As in TM systems, both software and hardware conflict detection mechanisms can be considered for supporting DRF*x*. We pursue a hardware detection mechanism since the required hardware logic is fairly simple and is similar to existing bounded hardware transactional memory (HTM) implementations such as Sun's Rock processor [Dice et al. 2009], Intel's Haswell [Intel Corporation 2012], and IBM's Blue Gene/Q [Haring et al. 2012]. A DRF*x* compiler can bound the number of memory bytes accessed in each region, enabling the hardware to perform conflict detection using finite resources. While small regions limit the scope of compiler and hardware optimizations, we discuss an approach in Section 6 that allows us to regain most of the

lost optimization potential. Specifically DRF$x$ hardware can execute and commit regions out-of-order, coalesce regions to reduce the number of conflict checks, and exploit temporal locality to exclude a significant fraction of accesses from participating in conflict detection. These optimizations significantly improve upon the performance overhead of the baseline hardware design for SC violation detection.

### 1.3. Contributions

This article makes the following contributions:

—We define the DRF$x$ memory model for concurrent programming languages via three simple and strong guarantees for programmers (Section 2). We also establish a set of conditions on a compiler and hardware design that are sufficient to provide these three guarantees.
—We present a formalization of the DRF$x$ memory model as well as of the compiler and hardware requirements (Section 3). We have proven these requirements are sufficient to enforce DRF$x$.
—We describe a detailed compiler and micro-architecture design that instantiates our approach (Section 5 and 6). In our design, the compiler ensures that regions have a bounded size, allowing a processor to detect conflicts using finite hardware resources. We describe a novel approach to regain most of the lost optimization potential due to small region sizes. The hardware detects conflicts *lazily*, and we describe several optimizations to our basic detection mechanism. We have implemented a DRF$x$-compliant C compiler by modifying LLVM compiler [Lattner and Adve 2004] and have built a simulator for the two DRF$x$-compliant hardware designs using the Simics-based FeS2 simulator [Neelakantam et al. 2008].
—We evaluate the performance cost of our compiler and hardware instantiations in terms of lost optimization opportunity for programs in the PARSEC and SPLASH-2 benchmark suites (Section 7). The results show that the performance overhead is on average 9.6% when compared to the baseline fully optimized implementation.
—We discuss the lessons we learned from this project, including limitations and challenges of our approach that we had not anticipated and the relationship to our work on ensuring SC for all programs.

## 2. MOTIVATION AND OVERVIEW

This section motivates the problem addressed in the article and provides an overview of our solution through a set of examples.

### 2.1. Data Races

Two memory accesses *conflict* if they access the same location and at least one of them is a write. A program state is *racy* if two different threads are about to execute conflicting memory accesses from that state. A program contains a data race (or simply a race) if it has a sequentially consistent execution that reaches a racy state. Consider the C++ example in Figure 1(a). After thread `t` executes the instruction `A`, the program enters a racy state in which thread `t` is about to write to `init` while thread `u` is about to read that same variable. Therefore the program contains a data race.

### 2.2. Compiler Transformations in the Presence of Races

It is well known that *sequentially valid* compiler transformations, which are correct when considered on a single thread in isolation, can change program behavior in the presence of data races [Adve and Hill 1990; Gharachorloo et al. 1990; Manson et al. 2005]. Consider the C++ example from Figure 1(a). Thread `t` uses a Boolean variable `init` to communicate to thread `u` that the object `x` is initialized. Note that although

```
X* x = null;
bool init = false;

// Thread t          // Thread u
A: x = new X();      C: if(init)
B: init = true;      D:   x->f++;
```

(a)

```
X* x = null;
bool init = false;

// Thread t          // Thread u
B: init = true;
                     C: if(init)
                     D:   x->f++;
A: x = new X();
```

(b)

Fig. 1. (a) Original program. (b) Transformed program.

```
X* x = null;
atomic<bool> init = false;

// Thread t          // Thread u
A: x = new X();      C: if(init)
B: init = true;      D:   x->f++;
```

Fig. 2. Correct, data-race-free version of program from Figure 1.

the program has a data race, the program will not incur a null dereference on any SC execution.

Consider a compiler optimization that transforms the program by reordering instructions `A` and `B` in thread `t`. This transformation is sequentially valid, since it reorders independent writes to two different memory locations. However, this reordering introduces a null dereference (and violates SC) in the interleaving shown in Figure 1(b).[1] The same problem can occur as a result of out-of-order execution at the hardware level.

To avoid SC violations, languages have adopted memory models based on the DRF0 model [Adve and Hill 1990]. Such models guarantee SC for programs that are free of data races. The data race in our example program can be eliminated by explicitly annotating the variable `init` as `atomic` (`volatile` in Java 5 and later). This annotation tells the compiler and hardware to treat all accesses to a variable as "synchronization." However, two reads to variables tagged with `volatile` annotation do not induce a partial order among themselves. As such, (many) compiler and hardware reorderings are restricted across these accesses, and concurrent conflicting accesses to such variables do not constitute a data race. As a result, the revised C++ program shown in Figure 2 is data-race free and its accesses cannot be reordered in a manner that violates SC.

### 2.3. Writing Race-Free Programs Is Hard

For racy programs, on the other hand, DRF0 models provide much weaker guarantees than SC. For example, the C++11 memory model [Boehm and Adve 2008] considers data races as errors akin to out-of-bounds array accesses and provides no semantics to racy programs. This approach requires that programmers write race-free programs in order to be able to meaningfully reason about their program's behavior. But races are a common flaw, and thus it is unacceptable to require a program be free of these bugs in order to reason about its behavior. As an example, consider the program in Figure 3 in which the programmer attempted to fix the data race in Figure 1(a) using locks. Unfortunately, the two threads use different locks, an error that is easy to make, especially in large software systems with multiple developers.

[1]Although this "optimization" may seem contrived, many compiler optimizations, for example common-subexpression elimination and loop-invariant code motion, can have the effect of reordering accesses to shared memory.

```
X* x = null;
bool init = false;

// Thread t          // Thread u
A: lock(L);          E: lock(M)
B: x = new X();      F: if(init)
C: init = true;      G:   x->f++;
D: unlock(L);        H: unlock(M)
```

Fig. 3. An incorrect attempt at fixing the program from Figure 1.

```
// Thread t          // Thread u

A: lock(L);
C: init = true;
                     E: lock(M)
                     F: if(init)
                     G:   x->f++;
                     H: unlock(M);
B: x = new X();
D: unlock(L);
```

(a)

```
// Thread t          // Thread u

A: lock(L);
C: init = true;
B: x = new X();
D: unlock(L);
                     E: lock(M)
                     F: if(init)
                     G:   x->f++;
                     H: unlock(M)
```

(b)

Fig. 4. A program with a data race may or may not exhibit SC behavior at runtime. (a) Interleaving that exposes the effect of a compiler/hardware reordering under relaxed memory model. (b) Interleaving that does not.

Unlike out-of-bounds array accesses, there is no comprehensive language or library support to avoid data-race errors in mainstream programming languages. Further, like other concurrency errors, data races are nondeterministic and can be difficult to trigger during testing. Even if a race is triggered during testing, it can manifest itself as an error in any number of ways, making debugging difficult. Finally, the interaction between data races and compiler/hardware transformation can be counter-intuitive to programmers, who naturally assume SC behavior when reasoning about their code.

### 2.4. Detecting Data Races Is Expensive

This problem with prior data-race-free models has led researchers to propose to detect and terminate executions that exhibit a data race in the program [Adve et al. 1991; Boehm 2009; Elmas et al. 2007]. Note that it is not sufficient to only detect executions that exhibit a strictly simultaneous data race. While the existence of such an execution implies the existence of a data race in the program, other executions can also suffer from SC violations. Figure 4(a) shows such an execution for the improperly synchronized code in Figure 3. When executing under a relaxed memory model, statements B and C can be reordered. The interleaving shown in Figure 4(a) suggests an execution where the racing accesses to `init` do not occur simultaneously, but non-SC behavior (null dereference upon executing statement G) can occur. The execution has a *happened-before* data race [Lamport 1978; Fidge 1991; Mattern 1989].

Unfortunately, precise dynamic data-race detection either incurs 8× or more performance overhead in software [Flanagan and Freund 2009] or incurs significant hardware complexity [Prvulovic and Torrelas 2003; Muzahid et al. 2009]. The cost is due to the need to build a happened-before graph [Lamport 1978] of the program's dynamic memory accesses in order to detect races. A pair of racy accesses can be executed arbitrarily "far" away from each other in the graph. This increases the overhead of

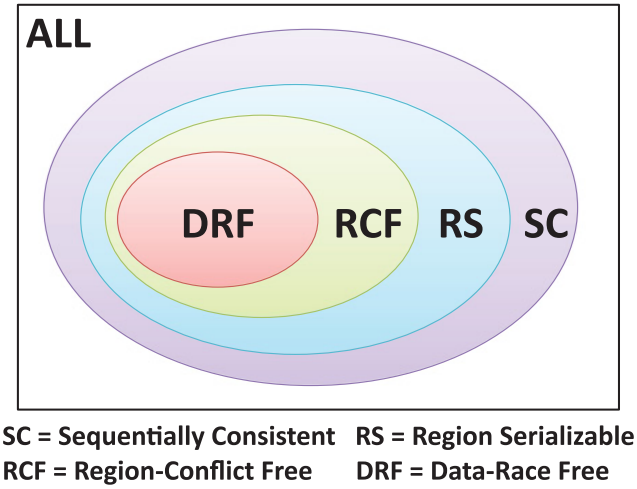


Fig. 5. The relationships among various properties of a program execution.

software-based detection and requires hardware-based detection to properly handle events like cache evictions, context switches, and so on. Imprecise race detectors can avoid some of these problems [Sen 2008; Marino et al. 2009a; Bond et al. 2010] but cannot guarantee to catch all SC violations, as required by the DRF$x$ memory model.

### 2.5. Detecting SC Violations Is Enough

Although implementing DRF$x$ requires detecting all races that may cause non-SC behavior, there are some races that do not violate SC [Gharachorloo and Gibbons 1991]. Thus, full happened-before race detection, while useful for debugging, is overly strong for simply ensuring executions are SC. For example, even though the interleaving in Figure 4(b) contains a happened-before data race, the execution does not result in a program error. The hardware guarantees that all the memory accesses issued while holding a lock are completed before the lock is released. Since the `unlock` at D completes before the `lock` at E, the execution is sequentially consistent even though the compiler reordered the instructions B and C. Therefore, the memory model can safely allow this execution to continue. On the other hand, executions like the one in Figure 4(a) do in fact violate SC and should be halted with a MM exception.

The Venn diagram in Figure 5 clarifies this argument (ignore the RCF and RS sets for now). SC represents the set of all executions that are sequentially consistent with respect to a program P. DRF is the set of executions that are data-race free. To satisfy the SC Soundness and Data-Race Completeness properties described in Section 1, all executions that are not in SC must be terminated and all executions in DRF must be accepted. However, the model allows flexibility for executions that are not in DRF but are in SC: It is acceptable to admit such executions since they are sequentially consistent, but it is also acceptable to terminate such executions since they are racy. This flexibility allows for a much more efficient detector than full-fledged race detection, as described below.

The DRF$x$ memory model only guarantees that non-SC executions *eventually* terminate with an exception. This allows SC detection to be performed *lazily*, thereby further reducing the conflict detector's complexity and overhead. Nevertheless, the Safety property described in Section 1 guarantees that an MM exception is thrown before the effects of a non-SC execution can reach any external component via a system call.

### 2.6. Enforcing the DRF$x$ Model

The key idea behind enforcing the DRF$x$ model is to partition a program into regions. Each region is a single-entry, multiple-exit portion of the program. Both the hardware and the compiler agree on the exact definition of these regions and perform program transformations only within a region. Each synchronization operation and each system call is required to be in its own region. For instance, one possible regionization for the program in Figure 3 would make each of {B,C} and {F,G} a region and put each lock and unlock operation in its own region.

During execution, the DRF$x$ runtime signals an MM exception if a conflict is detected between regions that are concurrently executing in different processors. We define two regions to conflict if there exists any instruction in one region that conflicts with any instruction in the other region. More precisely, we only need to signal an MM exception if the second of the two conflicting accesses executes before the first region completes. In the interleaving of Figure 4(b), no regions execute concurrently and thus the DRF$x$ runtime will not throw an exception, even though the execution contains a data race. On the other hand, in the interleaving shown in Figure 4(a), the conflicting regions {B,C} and {F,G} do execute concurrently, so an MM exception will be thrown.

### 2.7. From Region Conflicts to DRF$x$

The Venn diagram in Figure 5 illustrates the intuition for why the compiler and hardware co-design overviewed above satisfies the DRF$x$ properties. If a program execution is data-race free (DRF), then concurrent regions will never conflict during that execution, that is, the execution is *region-conflict free* (RCF), so an MM exception will never be raised. Since synchronization operations are in their own regions, this property holds *even in the presence of intra-region compiler and hardware optimizations*, as long as the optimizations do not introduce speculative reads or writes. This reasoning establishes the Data-Race Completeness property of the DRF$x$ model. Further, if an execution is RCF, then it is also *region-serializable* (RS): It is equivalent to an execution in which all regions execute in some global sequential order. That property in turn implies the execution is SC with respect to the original program. Again, this property holds even in the presence of non-speculative intra-region optimizations. This reasoning establishes the SC Soundness property of the DRF$x$ model.

In general, each of the sets illustrated in the Venn diagram is distinct: There exists some element in each set that is not in any subset. In some sense, this fact implies that the notion of region-conflict detection is *just right* to satisfy the two main DRF$x$ properties. On the one hand, it is possible for a racy program execution to nonetheless be region-conflict free. In that case, the execution is guaranteed to be SC, so there is no need to signal an MM exception. This situation was described above for the example in Figure 4(b). On the other hand, it is possible for an SC execution to have a concurrent region conflict and therefore trigger an MM exception. Although the execution is SC, it is nonetheless guaranteed to be racy. For example, consider again the program in Figure 3. Any execution in which instructions B and C are not reordered will be SC, but with the regionization described earlier some of these executions will trigger an MM exception.

### 2.8. The Compiler and the Hardware Contract

The compiler and hardware are allowed to perform any transformation within a region that is consistent with the single-thread semantics of the region, with one limitation: The set of memory locations read (written) by a region in the original program should be a superset of those read (written) by the compiled version of the region. This constraint ensures that an optimization cannot introduce a data race in an originally race-free

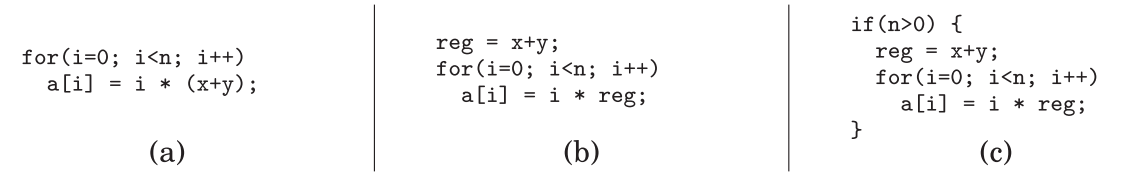


Fig. 6. A transformation that introduces new memory reads.

program. Note that a DRF0-compliant compiler is also forbidden from adding new memory writes into the compiled code, since that can introduce a race that changes program behavior. However, a DRF0-compliant compiler can add new memory reads, since such a race is harmless on non-race-detecting hardware [Boehm and Adve 2008].

Many traditional compiler optimizations (constant propagation, common subexpression elimination, dead-code elimination, etc.) satisfy the constraints above and are thus allowed by the DRF*x* model. Figure 6 describes an optimization that is disallowed by the DRF*x* model but allowed under DRF0. Figure 6(a) shows a loop that writes into each cell of an array. A transformation that allocates a register for the loop-invariant computation x+y is shown in Figure 6(b). However, on code paths in which the loop is never entered, this transformation introduces reads of x and y. While such behavior is harmless for sequential programs, it can introduce a race with another thread that modifies one of these variables. One way to avoid introducing reads is to explicitly check that the loop is executed at least once, as shown in Figure 6(c). The DRF*x* model allows the transformation with this modification, although our current compiler implementation simply disables the transformation. In spite of this, the experimental results in Section 7 indicate that the performance reduction due to lost compiler optimizations is reasonable, on average 6.2% on the evaluated benchmarks.

In addition to obeying the requirement above, the hardware is also responsible for detecting conflicts on concurrently executing regions. While performing conflict detection in software would avoid the need for special-purpose hardware, conflict detection in software can lead to unacceptable runtime overhead due to the need for extra computation on each memory access. On the other hand, performing conflict detection in hardware is efficient and lightweight, as demonstrated by the TM support in several existing processors [Dice et al. 2009; Intel Corporation 2012; Haring et al. 2012]. DRF*x* hardware can actually be simpler than TM hardware, since speculation support is not needed. Further, unlike in a TM system, the DRF*x* compiler can partition a program into regions of bounded size, thereby further reducing hardware complexity by safely allowing conflict detection to be performed with fixed-size hardware resources.

Having the compiler bound the size of regions is essential for efficient hardware detection, but the fences inserted by the compiler for the purposes of bounding should not unnecessarily disallow hardware optimizations. As such, the DRF*x* implementation supports two types of fences: hard fences that surround synchronization operations and system calls, and soft fences that are inserted only for the purposes of bounding region size. Both the implementation and the formalism account for the fact that the hardware can perform certain optimizations across soft fences that it must not perform across hard fences.

## 3. FORMAL DESCRIPTION OF DRF*x*

This section describes the formalization of the DRF*x* model. Preliminary notation and definitions are introduced in Section 3.1. A formal set of requirements sufficient to establish the DRF*x* guarantees is broken down into the responsibilities of the compiler and those of the execution environment. The execution environment implementation described in this section is hardware but could potentially be a software interpreter

or some combination of hardware and software. Section 3.2 formally presents the requirements that DRF$x$ places on the compiler and establishes two key lemmas relating a source program to the output of a DRF$x$-compliant compiler. In Section 3.3 the responsibilities of the execution environment are formalized and two important properties of a DRF$x$-compliant execution are established. Finally, Section 3.4 uses these results to establish the properties of the DRF$x$ model. Full proofs are omitted here, but the interested reader can find them in prior technical reports [Marino et al. 2009b; Singh et al. 2011b].

### 3.1. Preliminary Definitions

A program $P$ is a set of threads $T_1, T_2, \ldots, T_n$ where each thread is a sequence of deterministic instructions including:

—regular loads and stores (regular accesses)
—atomic loads and stores (atomic operations)
—branches and arithmetic operations on registers
—a special END instruction indicating the end of a thread's execution
—fence instructions (a hard fence HFENCE and a soft fence SFENCE) used only in compiled programs

Note that we assume the source language and target language are the same (actually the source language is a subset of the target language), so both source programs and compiled programs are represented in the same way. An argument extending the results to a high-level source language will be presented later.

We assume the semantics of our language is given in terms of how an instruction changes a machine state $M$ that contains shared global memory locations as well as a separate set of local registers for each thread. This semantics dictates how a thread's abstract execution proceeds. We write $(M, I) \longrightarrow_T (\hat{M}, \hat{I})$ to mean that executing instruction $I$ in machine state $M$ results in machine state $\hat{M}$ with $\hat{I}$ poised to execute next in thread $T$. We write $(M, I) \longrightarrow_T^* (\hat{M}, \hat{I})$ to indicate several steps of execution (transitive closure of above). Fence instructions behave as no-ops: $(M, \text{HFENCE}) \longrightarrow_T (M, I)$, where $I$ is the next instruction in program order in $T$ and similarly for SFENCE.

We extend the notion of a thread's abstract execution to a program by having execution proceed by choosing any thread and executing a single instruction from that thread. We write:

$$(M, \{I_1, \ldots, I_j, \ldots, I_n\}) \longrightarrow_P (\hat{M}, \{I_1, \ldots, \hat{I}_j, \ldots, I_n\})$$

if $(M, I_j) \longrightarrow_{T_j} (\hat{M}, \hat{I}_j)$. We call one or more of these steps a (partial) abstract sequential execution:

$$(M, \{I_1, \ldots, I_n\}) \longrightarrow_P^* (\hat{M}, \{\hat{I}_1, \ldots, \hat{I}_n\}).$$

We define a *behavior* to be a pair of machine states and denote it by $M_{\text{start}} \rightsquigarrow M_{\text{end}}$. Intuitively, we use behaviors to describe a starting machine state and a machine state that is arrived at after executing some or all of a program. The standard notion of *sequential consistency* can be phrased in terms of behaviors and abstract sequential executions.

*Definition* 1. $M_0 \rightsquigarrow M$ is a sequentially consistent behavior for a program $P$, or $M_0 \rightsquigarrow M$ is SC for $P$, if there exists an abstract sequential execution $(M_0, \{I_{10}, \ldots, I_{n0}\}) \longrightarrow_P^* (M, \{\text{END}, \ldots, \text{END}\})$ where each $I_{i0}$ is the first instruction in thread $T_i$. We say that $M_0 \rightsquigarrow M$ is a sequentially consistent partial behavior for $P$ if there is a partial abstract sequential execution $(M_0, \{I_{10}, \ldots, I_{n0}\}) \longrightarrow_P^* (M, \{I_1, \ldots, I_n\})$ where each $I_{i0}$ is the first instruction in thread $T_i$.

We say that two memory access instructions $u$ and $v$ conflict if they access the same memory location, at least one is a write, and at least one is a regular access. We say that a program has a *data race* if it has a partial abstract sequential execution where two conflicting accesses are ready to execute. More formally:

*Definition* 2. A program $P$ has a data race if for some $M_0$, $u$, $v$,

$$(M_0, \{I_{10}, \ldots, I_{n0}\}) \longrightarrow_P^* (M, \{I_1, \ldots, u, \ldots, v, \ldots, I_n\}),$$

where each $I_{i0}$ is the first instruction in thread $T_i$ and $u$ and $v$ are conflicting accesses. We shall say that such a partial abstract sequential execution exhibits a data race.

This canonical, formal definition captures the notion of a simultaneous data race in an abstract sequential execution. As discussed in Section 2.4, optimized, racy programs can yield results that are not sequentially consistent even on executions where a strict simultaneous data race does not occur. Such executions exhibit a happened-before data race, where conflicting accesses are not ordered by any atomic synchronization operations. In several of our proofs, we look for a happened-before data race in an execution and then rearrange the execution trace to generate a valid execution of the program that exhibits a simultaneous data race.

### 3.2. DRF$x$-Compliant Compilation

As described informally in Section 2, one of the responsibilities of a DRF$x$-compliant compiler is to divide a program into code regions that satisfy several requirements. We formally capture these requirements with the notion of a valid thread partition, introduced here.

A partition $Q$ of a thread $T$ is a set of disjoint, contiguous subsequences of $T$ that cover $T$. Call each of these subsequences a region. Regions will be denoted by the metavariable $R$.

*Definition* 3. A partition $Q$ is *valid* if:

—each atomic operation and END operation is in its own region
—each region has a single entry point (i.e., every branch has a target that is either in the same region or is the first instruction in another region)

We extend the notion of abstract execution of a thread from instructions to regions as follows. We write $(M, R) \longrightarrow_T (\hat{M}, \hat{R})$ if $(M, I_1) \longrightarrow_T \cdots \longrightarrow_T (\hat{M}, I_n)$, where

—$I_1$ is the first instruction in $R$,
—$I_2, \ldots, I_{n-1} \in R$,
—$I_k \neq I_1$ for each $2 \leq k < n$, and
—$I_n$ is the first instruction in region $\hat{R}$ (it is possible that $\hat{R} = R$).

For threads with valid partitions, $(M, R) \longrightarrow_T (\hat{M}, \hat{R})$ intuitively means that beginning with memory in state $M$, executing the instructions in $R$ in isolation will result in memory having state $\hat{M}$ and $T$ ready to execute the first instruction in region $\hat{R}$. Extending this to programs, an abstract region-sequential execution is one where a scheduler arbitrarily chooses a thread and executes a single region from that thread.

We can now formally introduce the notion of region serializability, which will be the key to establishing that an execution of the optimized, compiled program is sequentially consistent with respect to the source program. We define *region-serializable* behavior for a program $P$ in terms of an abstract region-sequential execution.

*Definition* 4. We say $M_0 \rightsquigarrow M$ is region-serializable behavior, or RS, for $P$ with respect to thread partitions $Q_i$ if there is an abstract region-sequential execution

$(M_0, \{R_{10}, \ldots, R_{n0}\}) \longrightarrow_P^* (M, \{R_1, \ldots, R_n\})$, where each $R_{i0}$ is the first region given by partition $Q_i$ for thread $T_i$.

Now let us introduce notation for the read and write sets for a region given a starting memory state. $\texttt{read}(M, R)$ is the set of locations read when executing $R$ in isolation starting from memory state $M$. $\texttt{write}(M, R)$ is defined similarly. Note that these are sets and not sequences.

We can now describe the requirements the DRF$x$ model places on a compiler. Consider a compilation $P \curvearrowright P'$ where each thread $T_i$ in $P$ is partitioned into some number, $m_i$, of regions by $Q_i$. So we have

$$P = \{T_1, \ldots, T_n\} = \{R_{11} \cdots R_{1m_1}, \ldots, R_{n1} \cdots R_{nm_n}\}.$$

Furthermore, the compiled program has the same number of threads and each is partitioned by some $Q'_i$ into the same number of regions as in the original program. So we have

$$P' = \{R'_{11} \cdots R'_{1m_1}, \ldots, R'_{n1} \cdots R'_{nm_n}\}.$$

We consider such a compilation to be DRF$x$-*compliant* if:

(C1) The partitions $Q_i$ and $Q'_i$ are valid.
(C2) For all $i, j, M$, we have $(M, R_{ij}) \longrightarrow_{T_i} (\hat{M}, R_{ik}) \iff (M, R'_{ij}) \longrightarrow_{T'_i} (\hat{M}, R'_{ik})$
(C3) For all $i, j, M$, we have $\texttt{read}(M, R_{ij}) \supseteq \texttt{read}(M, R'_{ij})$ and $\texttt{write}(M, R_{ij}) \supseteq \texttt{write}(M, R'_{ij})$
(C4) Each region $R'_{ij}$ in the compiled program contains exactly one fence operation and it is the first instruction. Each of the fences surrounding an atomic operation must be an HFENCE. The fence preceding an END operation also must be an HFENCE.

Intuitively, the above definition of a DRF$x$-compliant compilation requires that a DRF$x$-compliant compiler choose valid partitions for a program's threads, perform optimizations only within regions, maintain the read and write sets of each region, and introduce HFENCE and SFENCE instructions to demarcate region boundaries. These fence instructions communicate the thread partitions chosen by a DRF$x$-compliant compiler to the execution environment. In the next section, we will refer to these as the *fence-induced* thread partitions of a program.

We now state the two key lemmas we have proven for DRF$x$-compliant compilations.

LEMMA 1. *If $P \curvearrowright P'$ is a DRF$x$-compliant compilation and $M_0 \rightsquigarrow M$ is a region-serializable behavior for $P'$ with respect to its fence-induced thread partitions, then $M_0 \rightsquigarrow M$ is a (partial) sequentially consistent behavior for $P$.*

PROOF SKETCH. We can transform an abstract region-sequential execution of $P'$ to an abstract region-sequential execution of $P$ due to (C2). Clearly an abstract region-sequential execution qualifies as an abstract sequential execution. □

LEMMA 2. *If $P \curvearrowright P'$ is a DRF$x$-compliant compilation and $P'$ has a data race, then $P$ has a data race.*

PROOF SKETCH. Essentially, we take a partial abstract sequential (but not necessarily region-sequential) execution of $P'$ that exhibits a simultaneous data race, truncate it to the earliest happened-before data race, and reorder the truncated trace while maintaining program dependencies to achieve a trace of $P'$ with a region-sequential prefix and a suffix containing a simultaneous race. The ability to perform this reordering and achieve a region-sequential prefix relies critically on (C1), which insists that atomic accesses are in their own region. We can then use (C2) and (C3) to construct an

abstract sequential execution of $P$ exhibiting a (possibly different) data race from the racy execution of $P'$. □

Full proofs for the lemmas in this section can be found in Marino et al. [2009b].

### 3.3. DRF$x$-Compliant Execution

We now formally specify the requirements that the DRF$x$ model places on a machine executing a program. Note that the requirements for the execution environment are cleanly separated from those for the compiler. All references to a program in this section refer to a compiled, optimized program and not to the original source program.

Unlike compilers, machines do not typically perform optimizations that completely transform the input program. They more or less faithfully execute the instructions given to them, reordering instruction execution and locally speculating in order to avoid expensive stalls. This difference is reflected in our formal description of a relaxed execution which records traces of instructions from the input program along with memory ordering information. It is also crucial for our formalism to model the data race detection mechanism in order to prove both Data-Race Completeness (we must show that all detections truly indicate a racy program) and SC-Soundness (we must establish that the *lack of a detection* precludes certain dynamic memory orderings that would violate region serializability). Note that data-race detection is performed for each region, but just like a machine may have multiple instructions in flight and execute them out of order, a machine may also have multiple regions in flight and perform conflict detection for them out of order, subject to certain restrictions.

We will represent a (partial) *relaxed execution*, $E$, of a program as a 5-tuple $E = (M_0, \mathcal{T}, \text{EO}, \text{RCS}, err)$. Each of the components is described below:

—$M_0$ is the initial machine state

—$\mathcal{T}$ is a set of individual, dynamic thread traces ($\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$). Each thread trace $\tau_i$ contains all of the instructions executed so far in thread $i$ starting with the first instruction and continuing in program order without skipping over any instructions. (Note that any out-of-order instruction execution performed by the machine is not captured by these traces.) We call this order TO; it totally orders the dynamic instructions executed within a thread and is a partial order on all instructions in the program execution.

Each thread trace is divided into *dynamic regions* (notated using metavariable $\rho$), with all instructions between two fence instructions in the trace belonging to one dynamic region. This is referred to as the *fence-induced partition*. We call the fence-induced partition *valid* if all atomic operations are immediately surrounded by HFENCE instructions. Although, strictly speaking, TO is a relation on instructions, we will also use it to order dynamic regions within a thread trace.

—EO is a relation that specifies a partial order on memory accesses. If two operations $u$ and $v$ access the same memory location and at least one of them is a write, then either $u <_{\text{EO}} v$ or $v <_{\text{EO}} u$. Furthermore, two operations that do not access the same memory location are not related by EO. EO uniquely defines the write whose value each read sees (i.e., the most recent write to the same location in EO). Note that EO ∪ TO may contain cycles, so the relaxed orderings allowed by optimizations such as out-of-order execution and store buffers are captured by EO rather than by the thread traces.

—RCS is a map from dynamic regions to a conflict detection state in the set {*uncommitted*, *lagging*, *committed*}. Intuitively, RCS models a conflict detection mechanism that works on the fence-demarcated regions and moves them through three states as they execute, from *uncommitted*, possibly to *lagging*, and, finally, to *committed* when detection successfully completes with no region conflict found. A conflict detection mechanism may commit SFENCE bounded regions out of program

order, and a *lagging* region is one that has not yet committed but for which a later region has committed.

—*err* is either $\emptyset$ or a single element of EO, $u <_{\text{EO}} v$. Intuitively, a non-empty *err* will indicate a conflicting pair of accesses in concurrently executing regions which triggers an MM exception. An execution that has $err = \emptyset$ is called exception-free, while an execution where $err \neq \emptyset$ is called exceptional.

We further define a hardware dependence partial order, D, that captures intra-thread data and control dependencies. D is a subset of TO and orders a read before a TO-subsequent memory access if the value returned from the read is used to compute: the address accessed by the subsequent instruction, the value written by the subsequent instruction, or the condition used by an intervening branch.[2] D also orders writes to a location before TO-subsequent accesses to the same location.

We say that an execution $E = (M_0, \mathcal{T}, \text{EO}, \text{RCS}, err)$ is *well formed* for a program $P$ if all of the following conditions are met:

(WF1) Each thread trace $\tau_i$ represents a valid sequential execution of thread $i$ in $P$ given that each read sees the value written by the (unique) closest preceding write in EO. [Instruction semantics are respected.]

(WF2) EO is consistent with D in the sense that $u <_{\text{D}} v \implies \neg v <_{\text{EO}} u$. Furthermore, $\text{EO}_{|\text{wr}} \cup \text{D}$ must be acyclic, where $\text{EO}_{|\text{wr}}$ is the subset of EO containing only write-to-read (i.e., read-after-write) dependencies ($u <_{\text{EO}_{|\text{wr}}} v \iff u <_{\text{EO}} v \wedge u$ a write $\wedge$ $v$ a read). [Execution order respects intra-thread dependencies, and speculative writes are not visible to other threads.]

(WF3) A *committed* or *lagging* region never follows an *uncommitted* region in a thread trace. That is, if there is some $\rho$ such that $\text{RCS}(\rho) = \textit{uncommitted}$, then for all $\rho'$ such that $\rho <_{\text{TO}} \rho'$, $\text{RCS}(\rho') = \textit{uncommitted}$. [All *uncommitted* regions appear consecutively at the end of each thread trace.]

(WF4) A *lagging* region always has some *committed* region following it in its thread trace. That is, for all $\rho$ such that $\text{RCS}(\rho) = \textit{lagging}$, there exists some $\rho'$ such that $\rho <_{\text{TO}} \rho'$ and $\text{RCS}(\rho') = \textit{committed}$. [A *lagging* region always has a later region that committed out of order.]

(WF5) All regions preceding an HFENCE in a thread trace are *committed*. No thread trace contains an atomic access without an HFENCE immediately following it. [An HFENCE stalls until conflict detection for all prior regions completes.]

Intuitively, conditions (WF1) and (WF2) simply ensure that our machine correctly executes instructions and obeys intra-thread data and control dependencies. In particular, Condition WF2 prevents a machine from speculatively writing a value and making it visible to other threads before a read on which the write depends completes.[3]

Conditions (WF3) and (WF4) establish some basic conditions that we assume for a conflict detection mechanism. Multiple *uncommitted* regions may be in-flight in a

[2]Note that even "artificial" dependencies, where a computation uses a value read from memory in such a way that it does not actually influence the subsequent instruction (for instance, the read value is XOR'd with itself always resulting in 0), are included in D.

[3]It is interesting to note that the conditions for a DRF$x$-compliant compilation *do* allow optimizations that introduce a speculative write within a region, as long as they do not change the read and write sets of any abstract execution of the region. Even so, "out-of-thin-air" results due to such optimizations are prevented by either detecting a data race or ensuring region serializable behavior during execution of the compiled program. By prohibiting the hardware from making speculative writes visible to other threads, we facilitate our proof that if the hardware raises a data-race exception, then the program indeed has a data race. While we could likely relax this prohibition using something similar to the compiler's requirement to maintain read and write sets on all executions, this seems unnecessarily complex given that most hardware architectures already satisfy the condition.

thread simultaneously. Regions may commit out of order, but when this happens, prior *uncommitted* regions in the same thread must be classified as *lagging* regions. Condition (WF5) establishes that HFENCE instructions force all prior regions to commit. Furthermore, atomic operations may not complete (i.e., become visible to other threads) until their region is committed and the succeeding HFENCE is executed.

A well-formed execution has well-defined behavior and a conflict detection state that meets some basic structural conditions, but we have not yet specified what conflicts must be detected in order to establish the DRF$x$ guarantees. To be DRF$x$ compliant, an exception-free execution must exhibit region-serializable behavior, and an exceptional execution must imply a racy program. We have devised a set of conditions on the conflict detection mechanism that suffice to establish compliance. They essentially require a machine to commit a region only after all accesses in the region are complete and globally visible, and only if it can guarantee that the region's accesses either do not conflict with, or are memory-ordered before, any accesses in *uncommitted* regions on other threads. The conditions allow some flexibility to commit SFENCE-bounded regions out of order but ensure that memory ordering cycles cannot be introduced as a result of the *lagging* regions. Finally, the conditions require that the detection mechanism reports an exception only if there truly are conflicting accesses on different threads, neither of which is from a region that has already committed.

Formally, we call a well-formed execution $E = (M_0, \mathcal{T}, \text{EO}, \text{RCS}, err)$ DRF$x$ compliant if it satisfies all of the following conditions:

(E1) There exists a total order RO on committed and lagging regions that is consistent with EO $\cup$ TO lifted to regions. [The set of *committed* and *lagging* regions have an order consistent with thread and memory ordering.]
(E2) There do not exist a read $r \in \rho_r$ and a write $w \in \rho_w$ such that $\text{RCS}(\rho_r) \neq$ *uncommitted* and $\text{RCS}(\rho_w) =$ *uncommitted* and $w <_{\text{EO}} r$. [Reads in *committed* and *lagging* regions do not see writes in *uncommitted* regions.]
(E3) There do not exist a read $r \in \rho_r$ and a write $w \in \rho_w$ such that $\text{RCS}(\rho_r) =$ *uncommitted* and $\text{RCS}(\rho_w) =$ *committed* and $r <_{\text{EO}} w$. [Writes from *committed* regions are visible to reads in *uncommitted* regions.]
(E4) If $err = u <_{\text{EO}} v$, then $u \in \rho_u$ and $v \in \rho_v$ conflict, $u$ and $v$ are from different threads, neither $\rho_u$ or $\rho_v$ is *committed*, and at least one of these regions is *uncommitted*.

Intuitively, the conditions ensure a conflict detection mechanism in which *committed* regions are guaranteed not to contain accesses that participate in a race that violates region-serializability, while *lagging* regions are guaranteed to not participate in a race that violates region-serializability with accesses in other *lagging* regions or *committed* regions but may participate in a violating race with an access in an *uncommitted* region. Condition (E1) ensures that any race that would cause *committed* and *lagging* regions not to be serializable is caught. Condition (E2) requires that all reads in a region must complete before it or any subsequent region commits. Condition (E3) requires that all writes in a region must complete and be visible to other threads before it commits.

We need one more bit of notation to express our key lemmas: an operator on a well-formed, partial relaxed execution that truncates incomplete thread traces to include only *committed* and *lagging* regions. Note that Condition WF3 ensures that all *uncommitted* regions in a thread trace occur consecutively at the end. The truncation operator drops instructions from these *uncommitted* regions from the end of each trace, removes pairs from EO if at least one operation in the pair has been truncated from its thread trace, removes truncated regions from RCS, and sets *err* to $\emptyset$. We notate this as follows: $\lfloor (M_0, \mathcal{T}, \text{EO}, \text{RCS}, err) \rfloor = (M_0, \lfloor \mathcal{T} \rfloor, \lfloor \text{EO} \rfloor, \lfloor \text{RCS} \rfloor, \emptyset)$.

The following lemmas establish two key results for DRF$x$-compliant executions. Notice that these lemmas allow us to relate relaxed executions to properties that are

defined in terms of abstract, interleaved executions (region-serializable behavior and racy programs). In this way, they can be easily combined with Lemmas 1 and 2, which did not need to reason about relaxed execution ordering.

LEMMA 3. *If $E$ is a well-formed,* DRF$x$*-compliant execution of a (compiled) program $P$ with valid fence-induced thread partitions, then $\lfloor E \rfloor$ exhibits region-serializable behavior for $P$ w.r.t. to the fence-induced partitions.*

PROOF SKETCH. This follows quickly from (E1) and (E2). Condition (E2) establishes that any value read by an instruction in $\lfloor E \rfloor$ was written by an instruction that is also in $\lfloor E \rfloor$ and thus the truncated execution is well formed. Furthermore, Condition (E1) establishes an order on the regions in $\lfloor E \rfloor$ that is consistent with both thread order and the way that EO orders conflicting accesses within those regions. This establishes that the execution is serializable w.r.t. the regions. □

LEMMA 4. *If there is a well-formed, exceptional,* DRF$x$*-compliant execution of a (compiled) program $P$ with valid fence-induced thread partitions, then $P$ has a data race.*

PROOF SKETCH. From Lemma 3, we know that the execution has a region-serializable prefix. We then use this to construct an abstract sequential execution of the prefix. Because the execution is exceptional, Condition (E4) guarantees that we have conflicting accesses neither of which is contained in a *committed* region and at least one of which is from an *uncommitted* region. We can extend the execution of the prefix to an execution demonstrating a happened-before data race. Essentially, for a program with valid fence-induced thread partitions, a happened-before relation between operations on different threads implies the existence of an HFENCE following one operation on its thread and preceding the other on its thread. Since neither of the conflicting accesses is from a *committed* region, and Condition (WF5) requires regions preceding an HFENCE to be *committed*, we know the accesses cannot be related by happened-before. Finally, we derive from this an execution of $P$ that exhibits a simultaneous race.[4] □

Full proofs for the previous two lemmas can be found in Singh et al. [2011b]. Rather than starting with the conditions for well-formed, DRF$x$-compliant execution, the proofs in the cited technical report are done in the context of the particular architectural design described in Section 6. Conditions (E1) through (E4) capture the supporting lemmas from the technical report that are used to establish the results above.[5]

### 3.4. DRF$x$ Guarantees

Putting together the lemmas from Sections 3.2 and 3.3, we can prove the following theorem, which ensures that a DRF$x$-compliant compiler along with a DRF$x$-compliant execution environment enforce the SC Soundness and Race Completeness properties. We call an execution *complete* if either it is exceptional (contains a non-null *err* component) or all the thread traces in the execution terminate in an END operation.

THEOREM 1. *If $P \curvearrowright P'$ is a* DRF$x$*-compliant compilation, and $E$ is a complete* DRF$x$*-compliant execution of $P'$ with behavior $M_0 \rightsquigarrow M$, then either:*

—*$E$ is exception-free and $M_0 \rightsquigarrow M$ is sequentially consistent behavior for $P$*
or
—*$E$ is exceptional and $P$ contains a data race.*

[4]In fact, there are exceptional, DRF$x$-compliant executions where the conflict detected is not reachable through an abstract sequential execution, but this can only happen as the result of a previous data race that is reachable.

[5]Note that an earlier technical report [Marino et al. 2009b] establishes similar results under a different set of conditions that were too restrictive for the eventual hardware design.

The arguments presented above were developed entirely in the context of a low-level machine language. The results can, however, be extended to a high-level source language in the following way. Imagine a "canonical compiler" that translates each high-level statement into a series of low-level operations that read the operands from memory into registers, perform appropriate arithmetic operations on the registers, and then store results back to memory. Any optimizations are then applied after this canonical compiler is run. We can extend the results to the high-level language simply by requiring that the compiler choose a region partition that does not split up instructions that came from the same high-level source language expression or statement. This argument assumes that the number of memory accesses in the compilation of any statement in the source language is bounded by the maximum region size. If the source language does not guarantee this property, then its compiler can emit a warning in the rare case that a single source statement is forced to span multiple regions warning the user that, in the presence of data races, the statement may not execute atomically and an exception may not be thrown. We further discuss this and related issues in Section 8.3.

The definition of a DRF*x*-compliant execution and Lemma 3 establish that all DRF*x*-compliant executions are region-serializable up to the latest *committed* region in each thread. Combining this fact with Lemma 1, we can see that, restricted to *committed* and *lagging* regions, a DRF*x*-compliant execution is SC with respect to the original source program. Note that an HFENCE operation cannot execute until all previous regions in its thread are *committed* (condition (WF5)). Therefore, requiring that system calls are preceded by HFENCE instructions and only use thread-local data ensures that the behavior they exhibit would have been achievable in an SC execution of the original program. This establishes the Safety property of the DRF*x* model.[6]

## 4. COMPILER AND HARDWARE IMPLEMENTATION

There are several possible compiler and hardware designs that meet the requirements necessary to ensure the DRF*x* properties as described in the previous section. In the next two sections, we describe one concrete approach for the DRF*x*-compliant compiler and hardware. It is evaluated in the Section 7. The approach is based on two key ideas crucial for a simple hardware design:

—**Bounded regions:** First, the compiler bounds the size of each region in terms of number of memory accesses it can perform dynamically using a conservative static analysis. Bounding ensures that the hardware can perform conflict detection with *fixed-size* data structures. Detecting conflicts with unbounded regions in hardware would require complex mechanisms, such as falling back to software on resource overflow, that are likely to be inefficient.

—**Soft fences:** When splitting regions to guarantee boundedness, the compiler inserts a *soft* fence. Soft fences are distinguished from the fences used to demarcate synchronization operations and system calls that are called *hard* fences. While hard fences are necessary to respect the semantics of synchronization accesses and guarantee the properties of DRF*x*, soft fences merely convey to the hardware the region boundaries across which the compiler did not optimize. These smaller, soft-fence-delimited regions ensure that the hardware can soundly perform conflict detection with fixed-size resources. But it is in fact safe for the hardware to reorder instructions across soft fences whenever hardware resources are available, essentially erasing any hardware performance penalty due to the use of bounded-size regions.

[6]Condition E2 is also essential in establishing the Safety property since it ensures that no read preceding a system call sees a write from an *uncommitted* region that might not be part of an SC execution.

## 5. DRF$x$-COMPLIANT COMPILER

A DRF$x$-compliant compiler was built by modifying the LLVM compiler [Lattner and Adve 2004]. As specified by the requirements (C1) through (C4) in the previous section, to ensure the DRF$x$ properties, the compiler must simply partition the program into valid regions, optimize only within regions, avoid inserting speculative memory accesses, and insert fences at region boundaries.

### 5.1. Inserting Hard Fences for DRF$x$ Compliance

A hard fence is similar to a traditional fence instruction. The hardware ensures that prior instructions have committed before allowing subsequent instructions to execute and the compiler is disallowed from optimizing across them. To guarantee SC for race-free programs, the compiler must insert a hard fence before and after each synchronization access. On some architectures, the synchronization access itself can be translated to an instruction that has hard-fence semantics (e.g., the atomic `xchg` instruction in AMD64 and Intel64 [Boehm and Adve 2008]), obviating the need for additional fence instructions. In the current implementation, the compiler treats all calls to the `pthread` library and lock-prefixed memory operations as "atomic" accesses. In addition, since the LLVM compiler does not support the `atomic` keyword proposed in the new C++ standard, all `volatile` variables are treated as atomic. All other memory operations are treated as data accesses.

To guarantee DRF$x$'s Safety property, a DRF$x$-compliant compiler should also insert hard fences for each system call invocation, one before entering the kernel mode and another after exiting the kernel mode. Any state that could be read by the system call should first be copied into a thread-local data structure before the first hard fence is executed. This approach ensures that the external system can observe only portions of the execution state that are reachable in some SC execution. Transforming system calls in this way is not implemented in the compiler used for the experiments in Section 7.

To insert a hard fence, the compiler uses the `llvm.memory.barrier` intrinsic in LLVM with all ordering restrictions enabled. This ensures that the LLVM compiler passes do not reorder memory operations across the fence. LLVM's code generator translates this instruction to an `mfence` instruction in x86, which restricts hardware optimizations across the fence.

### 5.2. Inserting Soft Fences to Bound Regions

In addition to hard fences, the compiler inserts soft fences to bound the number of memory operations in any region. Soft fences are inserted using a newly created intrinsic instruction in LLVM that is compiled to a special x86 no-op instruction that can be recognized by the DRF$x$ hardware simulator as a soft fence. The compiler employs a simple and conservative static analysis to bound the number of memory operations in a region. While overly small regions do limit the scope of compiler optimizations, experiments show that the performance loss due to this limitation is about 6.2% on average (Section 7). After inserting all the hard fences described earlier, the compiler performs function inlining. Soft fences are then inserted in the inlined code. A soft fence is conservatively inserted before each function call and return and before each loop back-edge. Finally, the compiler inserts additional soft fences in a function body as necessary to bound region sizes. The compiler performs a conservative static analysis to ensure that no region contains more than $R$ memory operations, thereby bounding the number of bytes that can be accessed by any region. The constant $R$ is determined based on the size of hardware buffers provisioned for conflict detection.

The above algorithm prevents compiler optimizations across loop iterations, such as loop-invariant code motion, since a soft fence is inserted at each back-edge. However,

it would be possible to apply a transformation similar to loop tiling [Wolfe 1989] which would have the effect of placing a soft fence only once every $R/L$ iterations, where $L$ is the maximum number of memory operations in a single loop iteration. Restructuring loops in this way would allow the compiler to safely perform compiler optimizations across each block of $R/L$ iterations.

### 5.3. Compiler Optimization

After region boundaries have been determined, the compiler may perform its optimizations. By requirements (C2) and (C3), any sequentially valid optimization is allowed within a region, as long as it does not introduce any speculative reads or writes since they can cause false conflicts. As such, in the current implementation, all speculative optimizations in LLVM are explicitly disabled.[7] Note, however, that there are several useful speculative optimizations that have simple variants that would be allowed by the DRF*x* model. For example, instead of inserting a speculative read, the compiler could insert a special prefetch instruction that the hardware would not track for purposes of conflict detection. The Itanium ISA has support for such speculation [Triebel et al. 2001] in order to hide the memory latency of reads. Also, as shown earlier in Figure 6, loop-invariant code motion is allowed by the DRF*x* model, as long as the hoisted reads and writes are guarded to ensure that the loop body will be executed at least once and the loop block is contained in a region. As described in the previous section, regions could be constructed to have multiple iterations of a loop within a soft-fenced region, over which the compiler is able to perform loop-invariant code motion and other sequentially valid optimizations.

## 6. DRF*x*-COMPLIANT HARDWARE: DESIGN AND IMPLEMENTATION

This section discusses the proposed DRF*x* processor architecture. A lazy conflict detection scheme using bloom filter signatures is described, as well as several optimizations that allow efficient execution in spite of the small, bounded regions created by the DRF*x* compiler. We first give a brief overview of the design and then delve into more detail.

### 6.1. Overview

To satisfy DRF*x* properties, the runtime has to detect a conflict when region-serializability may be violated due to a data race and raise a memory model exception (Section 2.7). Figure 9 presents an overview of a DRF*x* hardware design that supports this conflict detection. Additions to the baseline DRF0 hardware are shaded in gray. The state of several hardware structures at some instant of time during an execution of a sample program is also shown. Section 6.11 discusses the implementation details of the proposed design.

In hardware transactional memory systems, the ability to rollback is a necessity. As such, they can easily tolerate false positives in their conflict detection mechanism by simply rolling back and re-executing. This allows them to use cache-line granularity conflict detection, which may report false races. DRF*x*, on the other hand, does not require a rollback mechanism. But, because it terminates an execution on detecting a race, false race reports cannot be tolerated. As such, DRF*x* performs byte-level conflict detection. Performing precise, *eager* byte-level conflict detection complicates the coherence protocol and cache architecture [Lucia et al. 2010]. For instance, such a scheme would require the hardware to maintain byte-level access state for every cache block, maintain the access state even after a cache block migrates from one processor to another, and clear the access state in remote processors when a region commits.

[7]The LLVM implementation has functions called `isSafeToSpeculativelyExecute`, `isSafeToLoadUnconditionally`, and `isSafeToMove`, which we modified to return `false` for both loads and stores.

Instead, DRF$x$ hardware employs lazy conflict detection [Hammond et al. 2004]. Each processor core has a *region buffer* that stores the physical addresses of all memory accesses executed in a region. An entry is created in the region buffer when a memory access is committed from the reorder buffer (ROB). We consider a load instruction *completed* when it *commits* from the ROB, while a store is considered *completed* when it *retires* from the store buffer. When all the memory accesses in a region have completed, the processor broadcasts the address set for the region to other processors for conflict checks. Once the requesting processor has received acknowledgments from all other processors indicating a lack of conflicts, it commits the region and reclaims the region buffer entries. The communication and conflict check overhead is reduced by using bloom-filter-based signatures to represent sets of addresses [Ceze et al. 2006]. Each processor core has a *signature buffer* that is used to store the read and write signatures for all its in-flight regions.

The region buffer has to be at least as large as the maximum number of instructions allowed to be executed in a soft-fenced region created by the DRF$x$ compiler. The static analysis used by the DRF$x$ compiler to guarantee this bound is necessarily conservative and may create regions that are much smaller than the desired bound. But performing frequent conflict checking for very small soft-fenced regions would hurt performance. To reduce this cost, our hardware coalesces adjacent regions separated by a soft fence into a single region at runtime when there is sufficient space available in the region buffer. Supporting this optimization requires using a region buffer somewhat larger than the maximum possible region-size guaranteed by the compiler.

When executing a hard fence, the DRF$x$ hardware stalls the execution of all future memory accesses until all accesses preceding the fence have completed. This helps guarantee correct behavior of synchronization operations and ensures that any conflicts that are detected indeed correspond to a data race. But it also prevents full utilization of processor resources since instruction and memory level parallelism cannot be exploited across the fence. If the more frequently occurring soft fences behaved the same as hard fences, then these lost opportunities to exploit parallelism would result in significant performance overhead. Fortunately, this is unnecessary since soft fences do not indicate the presence of synchronization. In fact, memory accesses from a region can be allowed to execute even if earlier regions that end in soft fences have not committed. In addition, regions separated by a soft fence can be committed out of order. The formal proofs outlined in Section 3 admit these optimizations and establish that the DRF$x$ runtime requirements are still satisfied.

### 6.2. Signature-Based Lazy Conflict Detection

Let us assume that a processor treats soft fences the same as hard fences, an assumption that we will relax later in the discussion. DRF$x$ hardware employs lazy conflict detection to detect when region-serializability could have been violated due to a data race.

Each processor core has a *region buffer* to record the addresses of memory locations accessed in a region. Each region buffer entry corresponds to a cache block and stores the cache block's address along with two bit vectors that keep track of which bytes in the cache block have been read and which have been written. This organization of the region buffer entry allows us to use a single region buffer entry to track multiple memory accesses to the same cache block within a region. The DRF$x$ compiler bounds the size of a soft-fenced region to a predefined bound B, which determines the minimum size that a processor needs to provision for a region buffer. In practice, however, most regions use fewer entries during execution since accesses to the same cache block are coalesced.

Similarly to DRF0 hardware, the memory accesses within a region can execute out-of-order, and in the case of stores, retire from a store buffer out-of-order. An entry in the region buffer is created for a memory access when it is committed from the ROB.

Once all of a region's memory accesses have committed from the ROB, and all of its stores have retired from the store buffer, the executing processor broadcasts the region's address set to the other processors to perform conflict checks. On receiving a conflict check request, a processor intersects the received address set with the addresses currently in its region buffer. If this intersection is empty, then an ACK message is sent to the requester indicating that no conflicts were detected. After receiving acknowledgments from all other processors, the requesting processor commits the region by deleting its address entries from the region buffer.

Broadcasting every address accessed by every region and checking their presence in every other processor's region buffer is clearly expensive. To reduce this cost, bloom-filter-based signatures [Ceze et al. 2006] can be used. We use two signatures to summarize the memory addresses accessed by a region, one for reads and one for writes. Signatures for each in-flight region are stored in the *signature buffer* (more than one region could be in-flight due to the out-of-order execution optimizations discussed later in Section 6.5). To perform conflict checks for a region, a processor first broadcasts only its read and write signatures. To conservatively detect whether two address sets summarized by bloom filter signatures might intersect simply requires an AND operation. Each processor checks for potential read-write or write-write conflicts by comparing the incoming signatures with each of the signatures currently in its signature buffer. If a potential conflict is detected, then a NACK is sent to the requester. Upon receiving a NACK, the processor sends the full address set for the region so precise conflict detection can be performed.

The size of our bloom filter signatures needs to be large enough so false conflicts, which require the expensive transmission of full address sets, are rare. On the other hand, large signatures could themselves incur significant communication overhead. But, since many regions can be in flight in a processor at once, the signature may be compared with many remote regions, increasing the probability of getting a false conflict. To address this problem, large signatures (1024 bits) are used, but they are compressed before transmission to reduce communication overhead. We use cache block addresses to build these signatures, helping to keep the number of unique addresses in each signature small. Because many regions have small access sets, their signatures are effectively compressed using a simple, efficient run-length encoding scheme. This strategy resulted in very high compression ratios that significantly reduced communication overhead.

### 6.3. Concurrent Region Conflict Check and Region Execution

When a processor P receives a conflict check request for a region R′, it need not stall the execution of its current region R while it performs the conflict check. A conflict check can be performed in parallel with the execution of a local region. The intuition here is that any memory access in the pipeline (and not yet included in the region and signature buffers) can be shown to have executed after the memory accesses in R′ because all accesses from R′ have already been completed. Thus, if no conflicts are detected, we can order R′ before R in the region serialization of the execution.

However, care must be taken to ensure that all memory accesses in the pipeline are actually completed after the memory accesses in R′. Stores are not completed before they are committed from store buffer, and therefore a store in the pipeline is definitely not completed. But an in-flight load may have already read its value from the cache while still in the pipeline before it has reached the head of the ROB and created an entry in the region buffer. To ensure that its value has not been changed by a completed

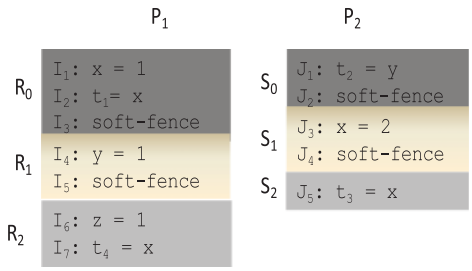


Fig. 7. An example binary compiled using a DRF$x$ compiler.

region that has been serialized before the current region R, any load that has already read the data from cache is re-executed if the accessed cache block is invalidated or evicted before the load is committed from the ROB. This mechanism is the same as the speculative load-execution technique proposed by Gharachorloo et al. [1991] to improve the performance of both SC and TSO hardware. The same mechanism can be used in DRF0 hardware to speculatively execute loads across fences. In our DRF$x$ design, this mechanism is enforced for all loads (even if they are not speculating across fences). We expect the overhead of this re-execution to be small since cache invalidations between instruction issue and retirement are rare.

### 6.4. Coalescing Soft-Fence-Bounded Regions

The DRF$x$ compiler uses a conservative static analysis to estimate the maximum number of instructions executed in a region. This could result in frequent soft fences. But a processor can dynamically ignore a soft fence if the preceding soft-fenced region executed fewer memory accesses than a predetermined threshold T. Combining two contiguous soft-fenced regions at runtime does not violate the DRF$x$ guarantees, because any conflict detected over the newly constructed larger region is possible only if there is a race, and ensuring serializability of the larger, coalesced soft-fenced regions is sufficient to guarantee SC for the original unoptimized program.

However, the processor needs to ensure that the newly constructed region does not exceed the size of its region buffer. The design guarantees this by using a region buffer that is of size T + B, where B is the compiler specified bound for a soft-fenced region, and T is the threshold used by a processor to determine when to ignore a soft fence. Too high a value for the threshold T would result in large regions at runtime, which might negatively impact performance due to the increased probability of false conflicts being detected in the bloom filter signatures. Also, it could undermine the out-of-order commit optimization discussed in Section 6.6.

### 6.5. Out-of-Order Execution of Regions

When a processor encounters a hard fence, it must wait for all memory accesses from preceding regions to complete before executing memory accesses from the later region. This is clearly a requirement for hard fences, since we may detect false data races if memory accesses are allowed to be reordered across hard fences that demarcate synchronization operations. However, this execution ordering can be relaxed for soft fences, allowing multiple uncommitted regions to be in-flight simultaneously. For example, in Figure 7, $I_7$ can be allowed to execute even if regions $R_0$ and $R_1$ have pending memory accesses in the ROB or the store buffer. If there is a pending store in a previous region (e.g., $I_1$), then its value can be forwarded to a load in a later region (e.g., $I_7$).

The correctness of the above optimization can be intuitively understood by observing that executing memory accesses out-of-order only results in more in-flight accesses that could potentially conflict. Therefore, it does not mask any conflicts that would

have been detected before. Also, reordering accesses across soft fences will not cause any access to be reordered across a synchronization operation. As such, any conflict that is detected as a result of this reordering still implies the presence of a data race.

### 6.6. Out-of-Order Commit of Regions

Once a region's memory accesses have completed, a processor can initiate a conflict check and commit the region from its region buffer if the check succeeds. Since instructions are committed from the ROB in program order, it is guaranteed that when a region is ready to commit, all memory accesses from the preceding regions would have also committed from the ROB. There could, however, be stores in the store buffer pending for the earlier regions. As a result, those earlier regions would not yet be ready to commit. In this scenario, it is possible to conflict check and commit the later region whose accesses have all completed. The not-yet-committed, prior regions correspond to the *lagging* regions in the formalism described in Section 3.3. In order to satisfy Conditions (E1) and (E2) for *lagging* regions, addresses for the uncommitted, previous regions must be included in the conflict check message for the later region.

For example, in Figure 7, say region $R_0$ is waiting for its store $I_1$ to be retired from the store buffer. In the meantime, $I_4$ has completed and has retired from the store buffer. Now $R_1$ is ready to commit. The processor can perform conflict check for $R_1$ (including the addresses from all prior uncommitted regions), and if no conflict is detected, commit it by deleting its entries from the region and signature buffers (but leaving the entries for uncommitted, prior regions). This optimization can be intuitively understood by observing that even if a write from $R_0$ lingering in the store buffer eventually causes a conflict with an access in another processor's pipeline, then the successful conflict check of the addresses in $R_1$ and $R_0$ at the time $R_1$ commits establishes a global order of all committed and lagging regions in the system at that point. This guarantees SC behavior up to the latest committed region in each thread.

### 6.7. Exploiting Locality in Memory Accesses

The hardware design discussed so far includes all the addresses accessed within a region when performing the conflict detection. In order to reduce the number of addresses that need to be conflict checked for a region, we propose an optimization that exploits the temporal locality exhibited by applications. Our insight is that once a processor core has conflict checked an address, it does not need to perform the same check again until it relinquishes the coherence permission for that address.

Once a cache block address has been conflict checked for reads, subsequent reads to this block can be treated as non-conflicting and excluded from conflict detection until the processor relinquishes its cache coherence permission for that block. Similarly, once a write is conflict checked, future reads and writes to the block can be excluded from conflict detection until permission to the cache line is downgraded. In order to distinguish the read and write cases, each cache block is extended with two additional *safe* bits:

—`read-safe`: If set, then it indicates that the cache block has been conflict checked for read-permission.
—`write-safe`: If set, then it indicates that the cache block has been conflict checked for read and write permission. Note that the write-safe bit also implies the read-safe property.

On a cache miss, a cache block is fetched with both bits being cleared. If a memory operation accesses a block that has both safe bits cleared, then it is added to the region buffer and its cache block address is added to the signature corresponding to current region. If a write is performed with write-safe bit in the cache block cleared, then the

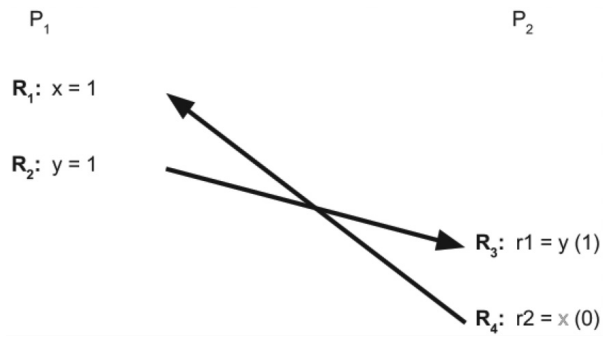


Fig. 8. Out-of-order commit and locality optimizations together can violate DRF$x$ guarantees.

write operation is added to the region buffer and its block address is included in the signature. In all other cases, the access is still added to the region buffer, but since the cache line has already been appropriately conflict checked, the cache block address is not included in the signature. In all cases, safe bits in the region buffer entry are updated based on the current state of the cache block's safe bits.

The cache block safe bits are updated during the conflict check that happens before region commit. During the first phase of the conflict check, the region's signatures are broadcast to all other cores. If there is no conflict detected in this first phase, then the read-safe and write-safe bits of the accessed blocks are set as appropriate: A cache block's write-safe bit is set if the region buffer indicates that the region performed a write access to the block and the read-safe bit is set if the region performed a read of the block.

If the second phase of conflict detection is required, then memory accesses that were found either read-safe or write-safe need not be sent unless the processor has since lost cache permission for the accessed block. This selective approach reduces the amount of data that are sent during the conflict detection process and reduces the bandwidth requirement imposed on the interconnect. However, if the processor core has lost coherence permission to a block, then we must be careful to ensure that accesses to this block are included in both phases of conflict detection.

To accomplish this, when a cache block with either its read-safe or write-safe bit set loses its coherence permission (eviction, invalidation, or downgrade from modified to a shared state), any region buffer entries matching the block are updated to clear the safe bits, and the block address is added to the appropriate region signatures based on the region buffer entries. In this way, the affected accesses will be included in both phases of conflict detection.

**Incompatibility with out-of-order region commit:** Since this design reduces demand on the interconnect and performs conflict detection only if there is a possibility of a conflict for memory accesses, it may be tempting to use this optimization along with out-of-order commit of regions optimization (Section 6.6). However, we observe that this locality optimization cannot be employed together with the out-of-order commit optimization, as it could result in violation the DRF$x$ guarantees. We illustrate this problem using the example shown in Figure 8. In this example, two processors ($P_1$ and $P_2$) are executing four soft-fenced transactions each containing only a single instruction. Let us assume that variables $x$ and $y$ are allocated on separate cache lines $C_x$ and $C_y$, respectively. Furthermore, assume that $C_x$ is cached by $P_2$ with `read-safe` bit set and the store operation in $R_1$ pending in the store buffer after missing in the cache. In the meantime $R_2$ completes its execution and starts its commit without waiting for the commit of preceding region. Since there are no conflicting accesses in $P_2$ (empty region

buffer), $R_2$ can be committed before $R_1$. Subsequently, $P_2$ executes $R_3$ and commits it without detecting any conflict as $R_2$ has been removed from the region buffer in $P_1$. After this, $R_4$ completes its execution. Note that the read operation will find the cache block with read-bit safe and will not participate in the conflict detection. Therefore, $P_2$ does not include $x$'s address in conflict detection and commits $R_4$ without detecting any conflict. Finally, $R_1$ completes and $P_1$ broadcasts $x$'s address during conflict detection, but it does not detect any conflict as $P_2$'s region buffer is empty at this point of time. Thus, $P_1$ commits $R_1$ without raising any MM exception. However, this execution is not SC because execution of the regions is not serializable. Therefore, out-of-order commit and cache locality-based optimizations are incompatible with each other in preserving DRF$x$ guarantees.

### 6.8. Handling Context Switches

A thread can incur a context switch at runtime for a variety of reasons. If a thread is context switched out in the middle of a region's execution, then we cannot detect SC violations caused by this partially executed region in which the compiler could have reordered memory accesses. Therefore, when possible, we require that the context switch be delayed until the subsequent soft-fence instruction. As our regions are bounded in the number of memory instructions, most well-behaved programs will eventually execute a soft fence after a finite amount of time. To account for problematic programs that perform unbounded computation while still performing a bounded number of memory accesses, we require that the DRF$x$ compiler insert additional soft fences in regions that could potentially execute unbounded number of instructions. By doing so, we make it possible for scheduler-induced thread context switches to be delayed without affecting the fairness of the operating system scheduler. For such delayed context switches, the hardware waits until all prior regions are committed and performs the context switch when the region buffer is empty.

Certain context switches, such as those induced by page faults and device interrupts, are critical and cannot be delayed. We observe that DRF$x$-style conflict detection should be disabled for low-level system operations such as the page-fault handler. It is unclear if halting such critical functionality with a memory-model exception is a good design choice. Instead, we propose that such low-level code be (either manually or statically) verified to be data-race free.

When critical context switches occur, the processor retains the region buffer entries for the switched-out thread. When the processor is executing the page-fault or the interrupt handler, it continues to perform conflict detection on behalf of the switched-out thread. Since conflict detection is disabled for the handler, no new entries are added to the region buffer. When the handler has finished, we require that the operating system schedule the switched-out thread on the same processor core. At this point, the thread continues using the region buffer, which contains the same entries it had at the time it was switched out.

While a page fault is being serviced for a thread, a processor can execute other threads instead of waiting for the data to be fetched from the disk. We can allow `N` context switches while handling a page-fault by provisioning a region buffer with a size that is `N` times that of the maximum bound specified by the compiler. For example, if the compiler bounds the region size to 64 locations and region buffer size is 512, we can allow eight context switches. Supporting multiple context per processor core would require detecting conflicts among threads concurrently executing on the same processor core. This can be done by virtualizing the region buffer by adding a context identifier. Adding a memory operation to the region buffer already requires searching it to determine if an entry for the cache block exists. During this search, the processor

can check if an entry with a conflicting access and a different context identifier exists, and, if so, raise a memory model exception indicating a data race.

### 6.9. Debugging Support

When a program is terminated with an MM exception, the processor provides the addresses of the starting and ending fence instructions of each conflicting region to assist in debugging.

A processor may encounter non-MM exceptions such as a null-pointer dereference, division by zero, and so on, while the current region is yet to complete. In this scenario, the processor stalls the execution of the current region and performs conflict detection for the partially executed region. If the conflict check succeeds, indicating no data race, then it raises the non-MM exception. But if a conflict is detected, then the processor throws an MM exception instead.

An MM exception in our design is imprecise in the sense that the state of the program when an exception is raised may not be SC, because the compiler or hardware could have already performed SC-violating optimizations in regions that contain racing accesses. Even an eager conflict detection scheme can only guarantee that the program state at the time of an exception is SC with respect to the compiled, binary program [Lucia et al. 2010]. The state could still be non-SC with respect to the source program due to compiler optimizations.

### 6.10. System Calls and Safety

The DRF$x$ compiler places each system call in its own region, separated from other regions by hard fences. Furthermore, the compiler generates code to ensure that system calls only access thread-local storage. Any user data potentially read by a system call is copied to thread-local storage before executing the preceding hard fence, and any user data written by the system call is copied out of thread-local storage after the succeeding hard fence.

An adversarial program may not obey the DRF$x$ compiler requirement that every region's size be bounded to a predefined limit. When a program executes a region that exceeds the bound, the DRF$x$ hardware can trivially detect that condition and raise an MM exception to ensure safety.

### 6.11. DRF$x$ Hardware Design Details

Region and signature buffers for each processor core are the main extensions to the baseline hardware structures. We assume a snoop-based architecture that we extend with additional messages to support conflict checking. Conflict check messages are independent of coherence messages. Figure 9 shows our DRF$x$ hardware extensions to a baseline out-of-order processor with store buffer. In the proposed design, we commit regions in-order but exploit the locality in memory accesses as discussed in Section 6.7. We now describe these extensions in detail.

When a hard fence is committed from the ROB, a new region is created by first finding a free entry in the signature buffer (one with its `valid` bit unset), initializing the entry's fields, and storing its index in the current Signature Buffer Index (SBI) register. The `SBI` register keeps track of the signature buffer entry corresponding to the region that is currently being executed. When a soft fence is committed from the ROB, we create a new region only if the current region size (stored in the `region-Size` field of the region's signature buffer entry) exceeds a pre-determined threshold `T` (32 in our experiments). If a hard or soft fence starts a new region, then its instruction address is stored in the `Region-beginPC` field of the new region's signature buffer entry. This information is used while reporting an MM exception.

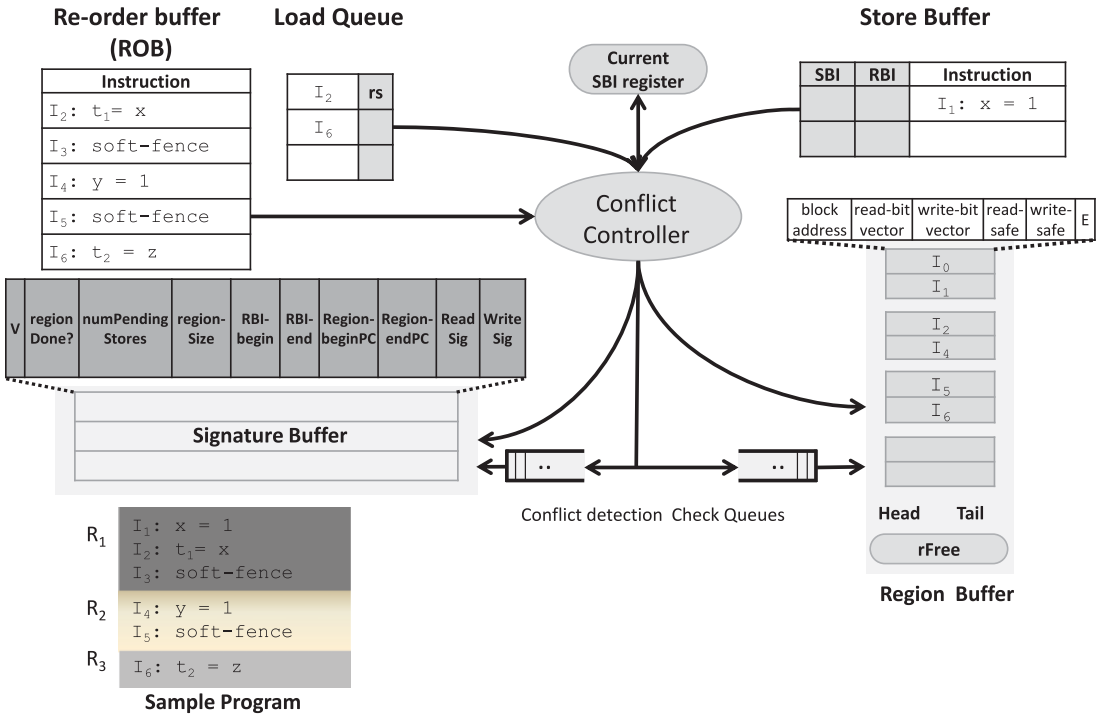


Fig. 9. Architecture support for DRF*x* (shown in gray).

When a memory instruction commits from the ROB, it searches the region buffer to check if there is an entry for the same cache block address already present for current region. A new region buffer entry is allocated for it if there is no prior entry for this cache block in the region buffer. The register `rFree` keeps track of the total number of free entries in the region buffer. If no free region entry is available, then the memory access is stalled at the commit stage of the pipeline. Free region buffer entries are organized as a free-list. The `head` and `tail` registers point to the first and last entries in the free list, respectively. The head and tail of a region is stored in the `RBI-begin` and `RBI-end` fields in the signature buffer entry of that region (RBI stands for Region Buffer Index).

A region buffer entry contains the following fields: cache block address, read and write bit vectors to keep track of bytes accessed within the cache block, `read-safe` and `write-safe` bits, and an `E` bit that is set when the cache block has lost its coherence permission. When a memory operation commits from the ROB, it sets the bits corresponding to the accessed memory location in the appropriate bit vector. Furthermore, for loads, the `read-safe` bit is updated based on the state of the safe bits in the cache when the block was accessed. For stores, the `write-safe` bits in the region buffer entry is updated on retirement (see below for details). The `read/write-safe` bits are reset when the corresponding cache block loses its coherence permission or is evicted from the cache. In any of these events, the *E* bit is also set in the entry to remember this event. When a region commits without detecting a conflict in signatures, it sets *safe* bits in the cache only if `E` bit in the region buffer entry is not set.

When a load reads from the cache, it needs to remember whether the accessed cache block had either of `read-safe` or `write-safe` bit set or not. To this end, we extend load queue entries to have additional bit: `rs`, which is set if either of `read-safe` or `write-safe` bits were set in the cache block. On a cache block invalidation, the load

queue is searched and the rs bits of any entries corresponding to the cache block being evicted are reset. When a load is committed from ROB, it uses the rs bit to update its region buffer entry. The address of the accessed cache block is inserted into read signature if the *rs* bit is not set. A store updates the write-safe bit in its region buffer entry when it retires from the store buffer. To update the region buffer entry directly, we save the RBI of the corresponding region buffer entry in the store buffer. It also updates the write signature if write-safe bit was not set in the cache.

In order to easily determine when all a region's stores have completed, a counter numPendingStores is kept in the region's signature buffer entry. This counter is incremented when a store is put into the store buffer and is decremented on retirement of a pending store from the store buffer.

When a hard or a soft fence commits from the ROB, its region's regionDone bit is set in the signature buffer. Also, its region's Region-endPC is updated with its instruction address. A region is ready to commit if its regionDone bit is set and numPendingStores is zero. Before committing a region, its addresses need to be conflict checked with all the in-flight regions in remote processor cores. During this process, the region's SBI is used as its identifier in the conflict check messages.

If the conflict check succeeds, then the region is committed by deallocating its entries in the signature and region buffers. The signature buffer entries are identified with the region's SBI used during conflict check. The start and end of a region's entries in the region buffer are determined using the RBI-begin and RBI-end fields stored in that region's signature buffer entry. These region buffer entries are then added to the free list.

### 6.12. Feasibility of DRF$x$ Hardware

In this section, we discuss the complexity cost of the proposed DRF$x$ hardware. The major changes include the addition of the region buffer, the signature buffer, and the conflict controller responsible for conflict detection.

Both region buffers and signature buffers can be efficiently implemented without adding significantly to the hardware complexity. The region buffer is organized similarly to a store buffer. However, instead of holding data values for memory accesses, each entry in the region buffer includes two small bit vectors. We only need a finite-sized region buffer because regions are statically bounded by the DRF$x$ compiler. The signature buffer in a processor core includes multiple bloom filters that can be efficiently implemented in the hardware [Ceze et al. 2006].

The major design complexity in DRF$x$ is due to the requirement of precise conflict detection. Our precise conflict detection mechanism involves two phases of conflict detection: conservative conflict detection via signatures and precise conflict based on the byte addresses of accessed memory locations. During the first phase of conflict detection, signatures are broadcast to other cores. Prior to a broadcast, signatures are compressed using a simple and efficient run-length encoding scheme. This scheme also allows a core to decompress signatures without incurring too much overhead.

During the first phase of conflict detection, two signatures are checked for a conflict. This check is efficiently performed by taking an intersection of two signatures using a bitwise AND operation. If the intersection is empty, then no conflict is detected for these two signatures. If a conflict is detected, then the second phase of conflict detection is required.

The second phase of conflict detection includes comparison of bit vectors associated with an entry in the region buffer. In this phase, a core receives a list of cache block addresses and associated bit vectors for read and write operations. Each address is sequentially searched in the region buffer by performing a CAM lookup. If a match is found, then bit vectors associated with the received cache block and matching region

Table I. Processor Configuration

| | |
|---|---|
| Processor | Four-core CMP. Each core operating at 2GHz. |
| Fetch/Exec/ Commit width | Four instructions (maximum 2 loads or 1 store) per cycle in each core. |
| Store Buffer | TSO: 64-entry FIFO buffer with 8-byte granularity.<br>DRF0, DRF*x*: 8-entry unordered coalescing buffer with 64-byte granularity. |
| L1 Cache | 64KB per-core (private), 4-way set associative, 64B block size, 2-cycle hit latency, write-back. |
| L2 Cache | 1MB private, 4-way set associative, 64B block size, 10-cycle hit latency. |
| Coherence | MOESI snoop protocol |
| Interconnection | Hierarchical switch, fan-out degree 4, 512-bit link width, 2-cycle link latency. |
| Memory | 80-cycle DRAM lookup latency. |
| Region buffer | 544 entry, 8 banks, 2-cycle CAM access. |
| Bloom filter | 1024 bits. 2 banks indexed by 9-bit field after address permutation [Ceze et al. 2006].<br>2-cycle access latency. |

buffer entries are compared for a match. These comparisons are performed by bitwise AND operations on bytemasks.

Another design complexity arises from how the proposed buffers (region and signature) are placed and routed on chip. This layout could impact the access latencies of these buffers. In our experiments we have assumed moderately aggressive latencies for these buffers.

## 7. PERFORMANCE EVALUATION

This section presents some performance results comparing the performance of programs compiled and executed under the DRF*x* memory model to those compiled and executed under a DRF0 model.

### 7.1. Methodology

The baseline compiler is the LLVM [Lattner and Adve 2004] compiler with all optimizations enabled (similar to compiling with the `-O3` flag in `gcc`) and with fences inserted before and after each call to a synchronization function and each access to a volatile variable.[8] The DRF*x* compiler is the implementation described in Section 5: hard fences are inserted before each call to a synchronization function and each access to a volatile variable, optimizations that perform speculative reads or writes are disabled, and soft fences are inserted to conservatively bound region size to 512 memory accesses.

Both the baseline and DRF*x* architectures are simulated using a cycle-accurate, execution-driven, Simics-based x86_64 simulator called FeS2 [Neelakantam et al. 2008]. The baseline architecture is a four-core chip multiprocessor operating at 2GHz (Table I). It allows both loads and stores to execute out of order between fences. The DRF*x* architecture adds support for soft fences and conflict detection as described in the previous section, using a region buffer of size 512 (compiler bound) + 32 (to support region coalescing).

Performance is measured over a subset of the PARSEC [Bienia et al. 2008] and SPLASH-2 [Woo et al. 1995] benchmarks. All of these benchmarks are run to completion. For PARSEC benchmarks (blackscholes, bodytrack, facesim, ferret, fluidanimate, streamcluster, swaptions), the `simmedium` input set was used. For SPLASH-2

[8]The unmodified LLVM compiler using its x86 backend targets hardware obeying the TSO memory model. The baseline simulated architecture uses a weaker memory model that permits additional reorderings not allowed by TSO. As such, we insert the additional fences around synchronization accesses to ensure that the program behaves correctly on the weaker model.

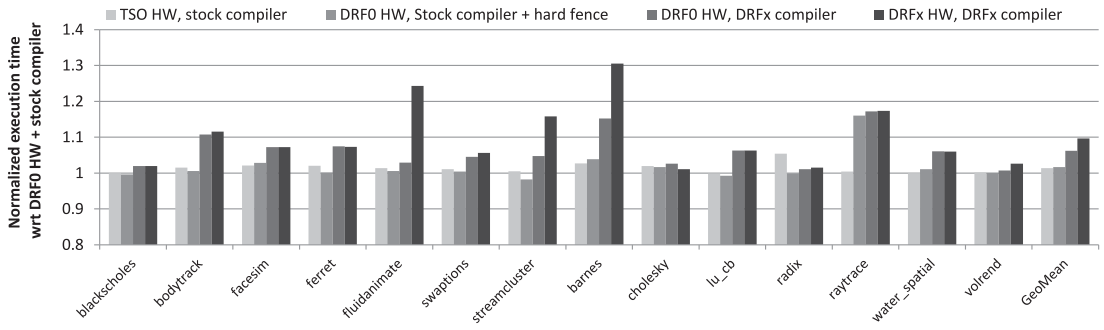


Fig. 10. Slowdown of benchmark programs run under the DRF$x$ model compared to a baseline DRF0 model, broken down in terms of cost of lost compiler optimization and cost of hardware race detection.

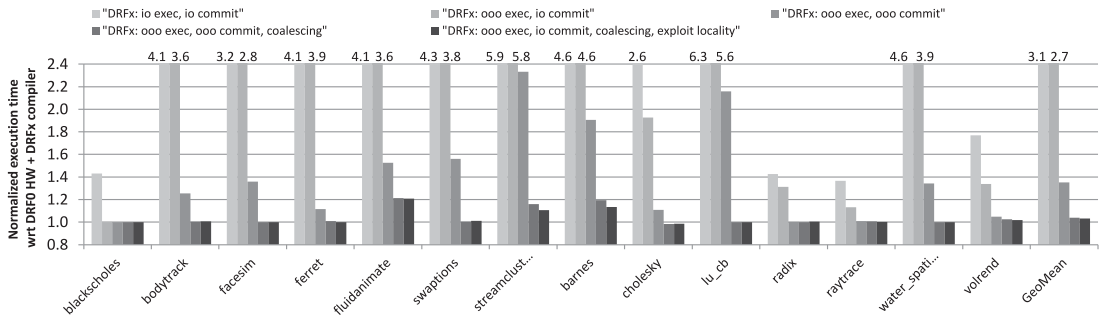


Fig. 11. Effectiveness of region coalescing and the out-of-order region execution and commit optimizations.

applications (barnes, cholesky, lu (contiguous), radix, raytrace, water-spatial, and volerand), the default inputs were used.

### 7.2. Comparison of DRF$x$ with Other Relaxed Memory Models

Figure 10 compares the performance of the TSO, DRF0, and DRF$x$ memory models. The results are normalized to the execution time of DRF0 hardware executing a binary produced by the stock LLVM compiler. Since the stock compiler is for x86 (TSO memory model), we added fences before and after synchronization operations to ensure correct memory ordering on DRF0 hardware. For DRF$x$, we measured both the cost of lost compiler optimizations and the cost of conflict detection in hardware. To measure the cost of lost compiler optimizations, we executed the binaries produced by the DRF$x$-compliant compiler on DRF0 hardware that treats soft fences as no-ops. To measure the cost of conflict detection, we evaluate a processor configuration that employs optimizations discussed in Section 6. In Figure 10, we find that a DRF$x$-compliant compiler (labeled as "DRF0 HW, DRF$x$ Compiler") incurs about 6.2% overhead on average due to restricted compiler optimizations. Conflict detection in the hardware adds about 3.4% overhead. In the following sections we will show that the optimizations proposed in Section 6 are crucial for low conflict detection overhead.

### 7.3. Effectiveness of DRF$x$ Hardware Optimizations

Figure 11 demonstrates the importance of distinguishing soft fences and implementing the optimizations described in Section 6. Here, performance is measured as execution time normalized to that of DRF0 hardware. When soft fences are treated like hard fences (label "DRF$x$: io exec, io commit"), the benchmarks experience slowdowns of more than 3× on average. Enabling out-of-order execution and commit for soft-fence-bounded regions significantly reduces this overhead to about 35.3% on average.

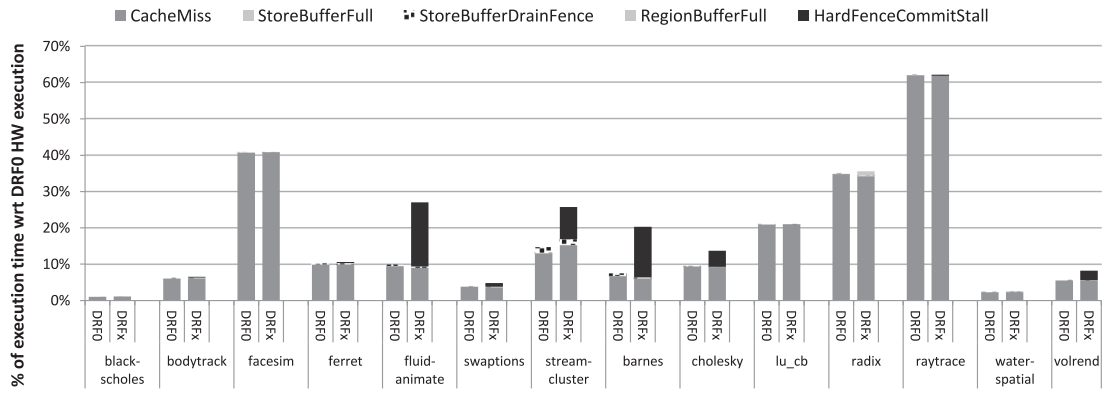


Fig. 12. Profile of commit stage stalls.

Coalescing soft-fenced regions further reduces this overhead to 4.0%. Coalescing of the soft-fenced regions is highly effective in increasing the average region size. The average soft-fenced region size without coalescing is only 8 memory operations. Enabling coalescing for the soft-fenced regions increases their average size to about 1,200 memory operations. This increase in region size leads to lower frequency of conflict detection and thus results in lower cost of the conflict detection. Finally, enabling the optimization for exploiting locality in memory accesses (Section 6.7) results in the overall best design and has about 3.1% performance overhead. Note that as mentioned earlier in Section 6.7, out-of-order commit is not compatible with the optimization for exploiting locality of memory accesses.

To understand different sources of overhead involved in the conflict detection, we profiled cycles during which commit from the ROB is stalled (Figure 12). Since entries are inserted into the region buffer at the commit stage, a core's commit could be stalled if the region buffer is full and cannot accept any new entry ("RegionBufferFull" in Figure 12). Furthermore, a hard fence in DRF*x* can introduce additional stalls due to the requirement that it must wait for all preceding regions (including the region immediately preceding the hard fence) to complete their conflict detection ("HardFenceCommitStall") and commit before it can be inserted in the region buffer. For some applications, conflict detection increases the stall cycles up to 17% of the total execution time of DRF0 hardware. The large overhead for `fluidanimate` is due to its usage of many locks (and hence hard fences). Furthermore, we find that the finite size of the region buffer has very limited impact on overall execution time ("RegionBufferFull" being almost negligible), primarily due to coalescing of the soft-fenced regions.

### 7.4. Scalability

Figure 13 shows the performance of DRF*x* as number of cores scale. As the number of cores increases, the broadcast latency and the rate of false conflicts in signatures also increase, leading to an increase in the cost of conflict detection. For the benchmarks `fluidanimate`, `barnes`, and `radix`, the increase in conflict detection overhead could be attributed to an increase in false conflicts when signatures are compared to a larger number of regions on more cores, whereas, for `streamcluster`, the average region size decreases with the increasing number of cores due to less work per thread and more frequent hard fences in the program. These smaller regions lead to an increase in the conflict detection overhead as the number of regions that a hard fence has to wait for also increases. For other benchmarks, we observe only a small increase in conflict detection overhead.

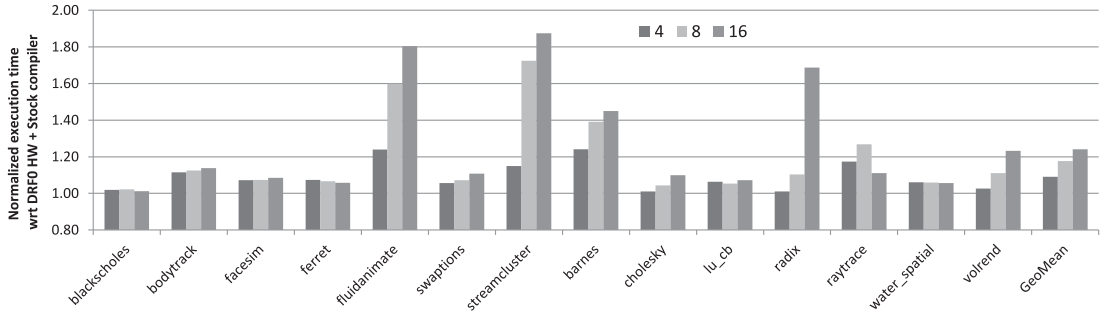


Fig. 13. Scalability of DRF$x$ with increasing number of cores.

## 8. CHALLENGES AND LESSONS LEARNED

### 8.1. Precise Conflict Detection

The biggest challenge for DRF$x$ is the need to efficiently perform precise conflict detection. In DRF$x$, we chose lazy conflict detection over eager as discussed in Section 6.1. In order to achieve precise conflict detection, DRF$x$ uses the region buffer and broadcasts a region's read/write sets before the region commits. This broadcast is one of the major challenges for a scalable DRF$x$ design. Bloom-filter-based signatures are effective in keeping the read/write set broadcast overhead low for a small number of cores. However, as we scale the number of cores, the probability of detecting a false conflict in signatures increases. While regions can be executed out-of-order to hide the overhead of conflict detection, this overhead cannot be hidden for hard fences, which must wait for conflict detection of earlier regions to complete. In Section 6.7, we described one approach to reduce the data sent during conflict detection, based on monitoring cache coherence permissions, but this also comes at a cost of increased hardware complexity.

The challenge of precise conflict detection was a major reason that we later pursed a "pessimistic" approach to ensuring sequential consistency, which simply disallows compiler and hardware optimizations that can violate SC [Marino et al. 2011; Singh et al. 2012]. To make this approach acceptable in terms of performance, both the compiler and hardware rely on information about which memory accesses are *safe*—accessing a thread-local or read-only location—and can therefore be reordered without violating SC. The compiler and hardware can aggressively optimize the safe accesses as usual but must conservatively handle unsafe accesses. The compiler uses a simple static analysis to determine these safe accesses, and the hardware leverages this information as well as dynamic information gleaned from the operating system's page protection mechanism.

### 8.2. SC vs DRF$x$

As described in Section 1 there are a complex set of tradeoffs between these two approaches to providing SC reasoning to programmers. First, it is unclear which model provides the best benefits in terms of programmability. The SC model is simpler because it ensures a strong guarantee for all program executions. On the other hand, the DRF$x$ memory model ensures region serializability for executions that do not raise exceptions, which is a stronger property than SC. In our current DRF$x$-compliant compiler, region boundaries are transparent to the programmer. However, one could imagine an implementation of DRF$x$ that makes the regions explicit (and others have pursued related ideas [Lucia et al. 2010; Sengupta et al. 2015]), thereby allowing programmers to reason about their code in terms of interleavings of atomic regions rather than individual instructions.

Second, it is unclear which model provides more opportunities for compiler optimizations. While the DRF*x* compiler is able to reorder all memory accesses within a region, the SC-preserving compiler can reorder memory accesses only if they are proven to be safe. However, the DRF*x* compiler must respect region boundaries and so has a smaller scope for its optimizations. Finally, it is unclear which model allows for the simplest hardware modifications. The DRF*x* hardware must perform precise conflict detection, which is a scalability challenge as described above. The SC hardware is comparatively simple, but it additionally relies on compiler and operating system extensions for identifying thread-local accesses.

### 8.3. Granularity Issues

As pointed out by Adve and Boehm [2010], an important issue in supporting SC for real-world programming languages, which we have ignored until now, is the need to define the granularity at which memory operations can interleave. A single line of source code, such as `i++` or `x = y`, can perform more than one memory access. Further, even a single read or write of an integer variable might translate into multiple memory accesses based on the alignment of the variable and the bit-width of the underlying architecture. While the hardware usually only provides atomicity of individual memory operations, it is desirable to specify the interleaving granularity at the programming language level that is independent of the underlying hardware.[9]

The DRF*x* memory model alleviates the granularity issue by guaranteeing atomicity of regions. Therefore, a programmer can assume that all statements in a region execute atomically on exception-free executions. The challenge, however, is in exposing the region boundaries in a meaningful way to the programmer. This is exacerbated by the fact that DRF*x* will sometimes be forced to insert a region boundary *within* a single source statement. For instance, an assignment to a structure might involve many memory accesses that exceed the size limitations of a region. Similarly, in a language like C++, the compiler can implicitly insert inlined functions, such as constructors, destructors, and overloaded operators, which may involve synchronization (e.g., those arising from memory allocation). Developing programming languages/development environments that expose these region boundaries to the programmer remains an unsolved problem.

In programs with no data races, the DRF0 model [Boehm and Adve 2008] guarantees that code regions that are free from synchronization operations are guaranteed to be atomic. Not surprisingly, both SC and DRF*x*, being strictly stronger memory models, equally enjoy the atomicity of such large regions for data-race-free programs.[10] DRF0 gets around the need to specify finer interleaving granularities as it provides no semantics for programs with data races. But data-race freedom is simply *assumed* rather than *enforced*. Thus, DRF0 does not enable programmers to safely rely on the atomicity of synchronization-free regions any more than under DRF*x* or SC. By providing a low-level possibly-hardware-specific interleaving granularity, SC provides a far stronger semantics than DRF0 for programs with data races. In contrast to SC and DRF0, DRF*x* is able to safely provide a larger interleaving granularity by dynamically checking for data races.

[9]Doing so for low-level languages like C and C++ has additional challenges. These languages allow the programmer to control the alignment of variables. But different architectures treat unaligned accesses differently—some generate a bus error while others do not.

[10]However, as pointed out above, there is still the problem of exposing synchronization operations carefully to the programmer. A single statement, such as `x = 17` is still not guaranteed to be atomic in the presence of compiler-inserted constructors or overloaded operators in DRF0, SC, and DRF*x*.

## 9. RELATED WORK

This section discusses the most closely related work.

### 9.1. Memory Models With Exceptions

The C++ memory model [Boehm and Adve 2008] and the Java memory model [Manson et al. 2005] are based on DRF0 [Adve and Gharachorloo 1996] and share its limitations for racy programs, which we discussed in Section 1.

Concurrently with our work on DRF$x$, Lucia et al. [2010] defined *conflict exceptions*, which also use a notion of regions to detect language-level SC violations in hardware. Their approach can be viewed as a realization of DRF$x$-compliant hardware, but it differs in important ways from our design. First, in their approach, a conflict exception is reported *precisely*, just before the second of the conflicting operations is to be executed.

Precise conflict detection is arguably complex in hardware as one has to track access state for each cache word and continue to track it even when a cache block migrates to a different processor core. Further, when a region commits, its access state needs to be cleared in remote processors. Finally, while this approach delivers a precise exception with respect to the binary, the exception is not guaranteed to be precise with respect to the original source program.

Second, in their approach, region boundaries are placed only around synchronization operations, thereby ensuring serializability of *maximal synchronization-free regions*, which is a stronger guarantee than SC. While this property could be useful for programmers, it can result in unbounded-size regions and thereby considerably complicates the hardware detection scheme and system software.

Adve et al. [1991] proposed to detect data races at runtime using hardware support. Elmas et al. [2007] augment the Java virtual machine to dynamically detect bytecode-level data races and raise a `DataRaceException`. Boehm [2009] provided an informal argument for integrating an efficient always-on data-race detector to extend the DRF0 model by throwing an exception on a data race. However, detecting data races either incurs 8× or more performance overhead in software [Flanagan and Freund 2009] or incurs significant hardware complexity [Prvulovic and Torrelas 2003; Muzahid et al. 2009]. A full data-race detector is inherently complex as it has to dynamically build the *happens-before* graph [Lamport 1978] to determine racy memory accesses. It is further complicated by the fact that racy accesses could be executed arbitrarily "far" away from each other in time, which implies the need for performing conflict detection across events like cache evictions, context switches, and so on. In contrast, DRF$x$ hardware is inherently simpler, as it requires that we track memory access state and perform conflict detection over only the uncommitted, bounded regions. Unlike earlier hardware data-race detectors that rely on post retirement speculation and checkpointing support, DRF$x$ does not require such complex hardware support. The primary complexity of post-retirement speculation and checkpointing arises from keeping track of speculative memory accesses even after their commit from the ROB. Furthermore, hardware data-race detectors like SigRace [Muzahid et al. 2009] use a signature-based conflict detection scheme (albeit an imprecise one), whereas the primary source of hardware design complexity for DRF$x$ is *precise* conflict detection, which is arguably simpler than post-retirement speculation and recovery.

Gharachorloo and Gibbons [1991] observed that it suffices to detect SC violations directly rather than data races. Their goal was to detect potential violations of SC due to a data race and report that to the programmer. However, their detection was with respect to the compiled version of a program. DRF$x$ incorporates the notion of compiler-constructed regions and allows the compiler and hardware to optimize within regions while still allowing us to dynamically detect potential SC violations at the language level.

More recently, Muzahid et al. [2012] and Qian et al. [2013] proposed hardware solutions to detect SC violation at the runtime. Instead of relying on data races as proxy SC violations, they record some metadata about memory accesses that complete before all earlier accesses have been completed and pass on this metadata to other processors with coherence messages to detect cyclic dependence chains. These solutions detect SC violations with respect to execution of a given binary and do not account for reorderings performed by the compiler. They are, however, inadequate for detecting conflicts among regions as region serializability is strictly stronger property than sequential consistency. In absence of the notion of regions, it is not clear how these solutions would account for memory reordering done by compiler. These solutions, therefore, are not sufficient for guaranteeing language-level SC.

### 9.2. Efficiently Supporting Sequential Consistency

If the hardware and the compiler can guarantee SC, then it is clearly preferable to weaker memory models. There have been several attempts to reduce the cost of supporting SC.

The Bulk compiler [Ahn et al. 2009] together with the BulkSC hardware [Ceze et al. 2007] provide support for guaranteeing SC at the language level. The bulk compiler constructs chunks similar to regions, but a chunk could span across synchronization accesses and could be unbounded. The BulkSC hardware employs speculation and recovery to ensure serializable execution of chunks. Conflicts are detected using a signature-based scheme and they are resolved through rollback and re-execution of chunks. Forward progress may not be possible in the presence of repeated rollbacks. The Bulk system addresses this issue and the unbounded chunk problem using several heuristics. When the heuristics fail, it resorts to serializing chunks and executing safer unoptimized code.

DRF*x* hardware could be simpler than Bulk hardware, as it avoids the need for speculation (especially across I/O) and unbounded region sizes that have been the two main issues in realizing a practical transactional memory system. However, DRF*x* requires precise conflict detection, whereas Bulk can afford false conflicts. Our observations that certain regions can execute and commit out-of-order and that conflict checks and region execution in different processors can all proceed in parallel is unique. It may help improve the efficiency and complexity of Bulk system as well.

SC can be guaranteed at the language level even on hardware that supports a weaker consistency model using static analysis to insert fences [Shasha and Snir 1988; Kamil et al. 2005; Sura et al. 2005]. However, computing a minimal set of fences for a program is NP-complete [Krishnamurthy and Yelick 1996]. One approach to reduce the number of fences is to statically determine potentially racy memory accesses [Kamil et al. 2005; Sura et al. 2005] and insert fences only for those accesses. These techniques are based on pointer alias analysis, sharing inference, and thread escape analysis. In spite of recent advances [Boyapati and Rinard 2001; Boyapati et al. 2002], a scalable and practically feasible technique for implementing a sound static data-race detector also remains an unsolved problem, as all the techniques require complex, whole-program analysis.

There has been much work on designing an efficient, sequentially consistent processor. But this only guarantees SC at the hardware level for the compiled program [Ranganathan et al. 1997; Blundell et al. 2009; Ceze et al. 2007; Wenisch et al. 2007; Lin et al. 2012].

More recently, we proposed an approach to guarantee end-to-end SC through separate modifications to the compiler and hardware [Marino et al. 2011; Singh et al. 2012]. The compiler is modified to be *SC-preserving*—to produce a binary that preserves SC if run on SC hardware—by restricting SC-violating optimizations to thread-local

variables. The hardware uses a combination of static and dynamic analysis to identify "safe" memory locations (both thread-local and shared, read-only data), which can be freely reordered, and uses an auxiliary store buffer to fast-track "safe" store commits. This approach requires neither whole-program compiler analysis [Kamil et al. 2005; Sura et al. 2005] nor complex checkpoint-and-rollback support in hardware [Ranganathan et al. 1997; Blundell et al. 2009; Ceze et al. 2007; Wenisch et al. 2007], resulting in an efficient and complexity-effective design. We have compared DRF$x$ against proposed SC-preserving compiler and hardware in Section 8.

### 9.3. Transactional Memory

HTM systems [Herlihy and Moss 1993] also employ conflict detection between concurrent regions. However, unlike TM systems, regions in DRF$x$ are constructed by the compiler and hence can be bounded. Also, on detecting a conflict, a region need not be rolled back. This avoids the complexity of a speculation mechanism. Thus, a DRF$x$ system does not suffer from the two issues that have been most problematic for practical adoption of TM.

Hammond et al. [2004] proposed a transactional coherency and consistency (TCC) memory model based on a transactional programming model [Herlihy and Moss 1993]. The programmer and the compiler ensure that every instruction is part of some transaction. The runtime guarantees serializability of transactions, which in turn guarantees SC at the language level. Unlike this approach, DRF$x$ is useful for any multi-threaded program written using common synchronization operations like locks, and it does not require additional programmer effort to construct regions. TCC also requires unbounded region and speculation support. TCC suggests that hardware could break large regions into smaller regions, but that could violate SC at the language level.

Our lazy conflict detection algorithm is similar to the one proposed by Hammond et al. [2004] but without the need for speculation and conflict detection over unbounded regions. Also, we employ signatures to reduce the cost of conflict checks. Unlike TM, DRF$x$ cannot afford false conflicts, which our design takes care to eliminate. But lazy conflict detectors like TCC assume some form of a commit arbiter to regulate concurrent commit requests for regions in different processors. As we discussed, we can allow all regions to be conflict checked in parallel with the execution of current regions, which could be simpler. Also, soft-fenced regions can be executed and committed out-of-order.

## 10. CONCLUSION

The DRF$x$ memory model for concurrent programming languages gives programmers simple, strong guarantees for all programs. Like prior data-race-free memory models, DRF$x$ guarantees that all executions of a race-free program will be sequentially consistent. However, while data-race-free models typically give weak or no guarantees for racy programs, DRF$x$ guarantees that the execution of a racy program will also be sequentially consistent as long as a memory model exception is not thrown. In this way, DRF$x$ guarantees safety and enables programmers to easily reason about *all* programs using the intuitive SC semantics. Furthermore, the minor restrictions DRF$x$ places on compiler optimizations are straightforward, allowing compiler writers to easily establish the correctness of their optimizations.

DRF$x$ capitalizes on the fact that sequentially valid compiler optimizations preserve SC as long as they do not interact with concurrent accesses on other threads. Since performing precise data-race detection is impractically slow in software and complex in hardware, DRF$x$ allows the compiler to specify code regions in which optimizations were performed. The hardware can then efficiently target data-race detection only at regions of code that execute concurrently. This allows the DRF$x$-compliant compiler and hardware to cooperate, terminating executions of racy programs that may violate

SC. The formal development establishes a set of requirements for the compiler and the hardware that are sufficient to obey the DRF*x* model. The implementation and evaluation indicate that a high-performance implementation of DRF*x* is possible.

While DRF*x* mostly hides the effect of compiler and hardware relaxations from the programmer, the state exposed to the program at a DRF*x* exception is non-SC. This poses interesting challenges on how a programmer might possibly recover from these exceptions. At one end, a solution would be to add in a hardware checkpointing support to undo the effects of currently executing regions before throwing an exception. At the other end, the compiler can bear the cost of recovery by emitting instructions to undo compiler optimization at each instructions. Exploring the resulting tradeoffs between compiler and hardware complexity is an interesting area for future work.

## REFERENCES

S. Adve and K. Gharachorloo. 1996. Shared memory consistency models: A tutorial. *Computer* 29, 12 (1996), 66–76.

Sarita V. Adve and Hans-J. Boehm. 2010. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM* 53, 8 (Aug. 2010), 90–101. DOI:http://dx.doi.org/10.1145/1787234.1787255

S. V. Adve and M. D. Hill. 1990. Weak ordering—A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ACM, 2–14.

S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. 1991. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*. 234–243.

Wonsun Ahn, Shanxiang Qi, Jae-Woo Lee, Marios Nicolaides, Xing Fang, Josep Torrellas, David Wong, and Samuel Midkiff. 2009. BulkCompiler: High-performance sequential consistency through cooperative compiler and hardware support. In *Proceedings of the 42nd International Symposium on Microarchitecture*.

C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

C. Blundell, M. M. K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*.

H. J. Boehm. 2009. Simple thread semantics require race detection. In *FIT Session at PLDI*.

H. J. Boehm and S. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 68–78.

Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 255–268. DOI:http://dx.doi.org/10.1145/1806596.1806626

C. Boyapati, R. Lee, and M. Rinard. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of OOPSLA*.

Chandrasekhar Boyapati and Martin Rinard. 2001. A parameterized type system for race-free java programs. In *Proceedings of OOPSLA*. ACM Press, 56–69.

L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. 2009. The case for system support for concurrency exceptions. In *USENIX HotPar*.

Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 278–289.

Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. IEEE Computer Society, 227–238.

D. Dice, Y. Lev, M. Moir, and D. Nussbaum. 2009. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of ASPLOS*.

T. Elmas, S. Qadeer, and S. Tasiran. 2007. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 2007 Conference on Programming Language Design and Implementation*. ACM, 245–255.

C. Fidge. 1991. Logical time in distributed computing systems. *IEEE Comput.* 24, 8 (Aug. 1991), 28–33. DOI:http://dx.doi.org/10.1109/2.84874

C. Flanagan and S. N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation*.

K. Gharachorloo and P. B. Gibbons. 1991. Detecting violations of sequential consistency. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'90)*. ACM New York, NY, USA, 316–326.

K. Gharachorloo, A. Gupta, and J. Hennessy. 1991. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the International Conference on Parallel Processing*. 355–364.

K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*. 15–26.

Lance Hammond, Vicky Wong, Michael K. Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. 102–113.

R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, G. L.-T. Chiu, P. A. Boyle, N. H. Chist, and Changhoan Kim. 2012. The IBM blue gene/Q compute chip. *IEEE Micro* 32, 2 (2012), 48–60. DOI:http://dx.doi.org/10.1109/MM.2011.108

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM, 289–300.

Intel Corporation. 2012. Intel architecture instruction set extensions programming reference. *319433-012 Edition* (Feb. 2012).

A. Kamil, J. Su, and K. Yelick. 2005. Making sequential consistency practical in titanium. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 15.

A. Krishnamurthy and K. Yelick. 1996. Analyses and optimizations for shared address space programs. *J. Parallel Distrib. Comput.* 38, 2 (1996), 130–144.

L. Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

L. Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* 100, 28 (1979), 690–691.

C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society.

Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. 2012. Efficient sequential consistency via conflict ordering. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans Boehm. 2010. Conflict exceptions: Providing simple parallel language semantics with precise hardware exceptions. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*.

J. Manson, W. Pugh, and S. Adve. 2005. The java memory model. In *Proceedings of POPL*. ACM, 378–391.

D. Marino, M. Musuvathi, and S. Narayanasamy. 2009a. LiteRace: Effective sampling for lightweight data-race detection. (2009).

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2009b. *DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages*. Technical Report 090021. UCLA Computer Science Department. http://fmdb.cs.ucla.edu/Treports/090021.pdf.

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFx: A simple and efficient memory model for concurrent programming languages. In *PLDI'10*. ACM, 351–362.

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A case for an SC-preserving compiler. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Friedemann Mattern. 1989. Virtual time and global states of distributed systems. In *Proceedings Workshop on Parallel and Distributed Algorithms*, Cosnard M. et al. (Ed.). North-Holland/Elsevier, 215–226. (Reprinted in: Z. Yang, T. A. Marsland (Eds.), *Global States and Time in Distributed Systems*, IEEE, 1994, pp. 123–133.).

Abdullah Muzahid, Shanxiang Qi, and Josep Torrellas. 2012. Vulcan: Hardware support for detecting sequential consistency violations dynamically. In *Proceedings of the 2012 45th Annual IEEE/ACM*

*International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, Washington, DC, USA, 363–375. DOI:http://dx.doi.org/10.1109/MICRO.2012.41

A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. 2009. SigRace: Signature-based data race detection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*.

N. Neelakantam, C. Blundell, J. Devietti, M. Martin, and C. Zilles. 2008. FeS2: A Full-system Execution-driven Simulator for x86. In *Poster at Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*.

M. Prvulovic and J. Torrelas. 2003. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. San Diego, CA.

Xuehai Qian, Josep Torrellas, Benjamin Sahelices, and Depei Qian. 2013. Volition: Scalable and precise sequential consistency violation detection. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 535–548. DOI:http://dx.doi.org/10.1145/2451116.2451174

P. Ranganathan, V. S. Pai, and S. V. Adve. 1997. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*. 199–210.

Koushik Sen. 2008. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 11–21. DOI:http://dx.doi.org/10.1145/1375581.1375584

Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. 2015. Hybrid static–dynamic analysis for statically bounded region serializability. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 561–575. DOI:http://dx.doi.org/10.1145/2694344.2694379

D. Shasha and M. Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (1988), 282–312.

Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi. 2011a. Efficient processor support for DRFx, a memory model with exceptions. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, 53–66.

Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madanlal Musuvathi. 2011b. *Efficient Processor Support for DRFx, a Memory Model with Exceptions*. Technical Report 110002. UCLA Computer Science Department. Retrieved from http://fmdb.cs.ucla.edu/Treports/110002.pdf.

Abhayendra Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. 2012. End-to-end sequential consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*. 524–535.

Z. Sura, X. Fang, C. L. Wong, S. P. Midkiff, J. Lee, and D. Padua. 2005. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2–13.

Walter Triebel, Joseph Bissell, and Rick Booth. 2001. *Programming Itaniumö-based Systems*. Intel Press.

Thomas F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. 2007. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*.

M. Wolfe. 1989. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing'89)*. ACM, New York, NY, 655–664. DOI:http://dx.doi.org/10.1145/76263.76337

S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 24–36.

Received July 2013; revised February 2016; accepted April 2016