Efficient Processor Support for DRFx, a Memory Model with Exceptions

Abhayendra Singh* Daniel Marino† Satish Narayanasamy* Todd Millstein† Madanlal Musuvathi‡

*University of Michigan, Ann Arbor †University of California, Los Angeles ‡Microsoft Research, Redmond

Abstract

A longstanding challenge of shared-memory concurrency is to provide a memory model that allows for efficient implementation while providing strong and simple guarantees to programmers. The C++0x and Java memory models admit a wide variety of compiler and hardware optimizations and provide sequentially consistent (SC) semantics for data-race-free programs. However, they either do not provide any semantics (C++0x) or provide a hard-to-understand semantics (Java) for racy programs, compromising the safety and debuggability of such programs.

In earlier work we proposed the DRFx memory model, which addresses this problem by dynamically detecting potential violations of SC due to the interaction of compiler or hardware optimizations with data races and halting execution upon detection. In this paper, we present a detailed micro-architecture design for supporting the DRFx memory model, formalize the design and prove its correctness, and evaluate the design using a hardware simulator. We describe a set of DRFx-compliant complexity-effective optimizations which allow us to attain performance close to that of TSO (Total Store Model) and DRF0 while providing strong guarantees for all programs.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: Hardware/software interfaces

General Terms Design, Performance

Keywords memory models, data-races, memory model exception, soft fences

1. Introduction

A memory consistency model (or simply a *memory model*) forms the foundation of shared-memory multi-threaded programming. It defines the order in which memory operations performed by one thread become visible to other threads. A programmer needs to understand the memory model in order to determine the behavior of a concurrent program. Every layer in the compute stack has to ensure that any transformations it performs retain the semantics of the source program with respect to the memory model.

Sequential consistency (SC) [27] is the most intuitive memory model for programmers. SC guarantees that all the memory operations executed across different threads appear in one total order that is consistent with the per-thread program order. Many compiler and hardware optimizations that are valid for sequential programs (such as register promotion, common subexpression elimination, out-of-order execution, store buffers, etc.) can break the SC semantics of concurrent programs. The resulting performance limitation has forced modern hardware architectures and mainstream programming languages to support weaker memory models.

In particular, recently proposed memory models for C++0x [9] and Java [30] are based on the data-race-free-0 (DRF0) memory model [1]. These models require that a programmer explicitly classify shared-memory variables into *synchronization* and *data*.[1] The intention is that programmers use synchronization variables for *all* inter-thread communication and that there are no data races on data variables. In other words, two threads in the program are not allowed to concurrently access the same data variable unless both accesses are reads. For such data-race-free programs, DRF0 guarantees SC. To provide this guarantee, DRF0 requires the language runtime to execute synchronization accesses in SC order with few optimizations. On the other hand, the runtime is allowed to perform most *sequentially valid* (i.e., correct on a single thread in isolation) optimizations in synchronization-free regions of the program. In this way DRF0 can be efficient and at the same time provide strong guarantees for data-race-free programs.

However, DRF0-based memory models provide weak or no guarantees for programs with data races. For instance, C++0x gives no semantics to racy programs. As a result, compiler and hardware optimizations can introduce arbitrary behavior for a racy program, potentially compromising the program's correctness and security properties. As this situation is unacceptable for type-safe languages, the Java memory model (JMM) provides a weak semantics for racy programs that precludes a class of adversarial behaviors. This semantics is subtle and complex and therefore makes understanding and debugging multithreaded program executions extremely difficult. It also makes it difficult to develop compiler and hardware optimizations that properly respect the JMM [13, 38].

Motivated by this difficulty, prior work [8, 16, 17, 19, 29, 32] has proposed a *fail-stop* semantics for racy programs. The basic idea is to treat data races as program errors and dynamically terminate racy executions with a *memory-model (MM) exception*. Doing so ensures that programmers are not exposed to the obscure effects of weak memory models by guaranteeing that executions not terminated with an MM exception are SC.

We recently developed a memory model based on this approach called DRFx [32]. Our key observation is that full data-race detection is unnecessary for the purpose of providing strong memory-model guarantees to programmers. For example, it is acceptable to allow a racy program to execute without raising an MM exception, as long as the execution is guaranteed to be SC. Accordingly, the

[1] Synchronization variables are annotated with type qualifiers such as `volatile` (in Java) or `atomic` (in C++). Unannotated variables are data variables.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0266-1/11/03. . . $10.00

DRFx memory model provides two simple and strong properties to programmers. First, a program execution that is not terminated with an MM exception is SC. Second, a program execution that is terminated with an MM exception has a data race. We presented a design for a compiler and runtime system that ensures these properties. The compiler partitions the program into *regions* and communicates region boundaries to the hardware via memory fences. The compiler and hardware may perform most sequentially valid optimizations within a region but may not optimize across region boundaries. Therefore, the DRFx properties can be ensured by dynamically detecting conflicting memory accesses between concurrently executing regions [29, 32]. If no such conflicts arise, then the execution is *region-serializable* which implies SC; otherwise the detected conflict indicates the presence of a data race in the program.

In this paper, we propose a detailed micro-architecture design for supporting the DRFx memory model. Our approach builds on a hardware design sketched in our prior work [32] but extends it with several important optimizations. Further, we have carefully formalized the approach, including these optimizations, at the micro-architectural level and have proven that it ensures the DRFx properties. Finally, we have built a simulator for our proposed hardware which, in conjunction with a DRFx-compliant compiler [32], allows us to perform an end-to-end performance comparison with the DRF0 memory model.

We have made a set of design choices in order to keep the hardware support for DRFx simple while still allowing optimizations for efficiency, both of which are critically important for future acceptance by processor vendors. First, we propose a *lazy* conflict detection scheme for DRFx that does not change the cache architecture or the coherence mechanism of the baseline DRF0 hardware. Unlike transactional memory (TM) support [23], DRFx cannot tolerate false conflicts arising either from regions accessing different words in the same cache line or from accesses occurring in mispredicted branches. Rather than modifying the cache coherence mechanism to track a region's access information at word granularity [29], we instead store the addresses accessed by a region in a dedicated *region buffer*. To commit a region, a processor sends the accessed addresses to other cores for a conflict check. While lazy conflict detection can delay an MM exception beyond the first instruction that results in a non-SC state, our implementation ensures that this state is not observable to the rest of the system by throwing the exception before any subsequent system call.

Second, performing conflict detection over unbounded-size regions would tremendously increase the complexity of the hardware support needed, due to the possibility of overflowing the region buffer. Indeed, this issue has been one of the main impediments to TM support in hardware. We observe that, unlike a TM system, the DRFx compiler has the flexibility to choose region boundaries and can therefore bound the number of memory accesses in each region in order to allow the hardware to perform conflict detection with finite resources. We employ a simple and conservative static analysis for this purpose.

While bounded regions simplify conflict detection, they can also lead to significant performance loss by limiting the instruction and memory parallelism available to the processor. To solve this problem, the DRFx compiler generates two kinds of fences to demarcate regions. *Hard* fences behave like traditional memory fences and are introduced at synchronization operations in order to preserve the programmer-intended ordering semantics. In contrast, *soft* fences indicate region boundaries introduced solely to bound the number of memory accesses per region. We propose and validate the following optimizations involving soft fences. First, a processor can execute, and even commit, soft-fenced regions out-of-order while still properly detecting conflicts as required by DRFx. Second, contiguous soft-fenced regions executed in a processor can be coalesced at runtime to form larger regions if space is available in the region buffer, which reduces the frequency of conflict checks. Finally, the conflict checks for regions in different processors can proceed in parallel, and a processor can continue to execute instructions while concurrently servicing conflict check requests from remote processors.

We have built a DRFx-compliant hardware simulator using the Simics-based FeS2 simulator [34]. In conjunction with a DRFx-compliant C compiler [32] built on top of LLVM [28], we compared the performance of DRFx to that of DRF0 compiler and hardware. For additional reference, we also evaluated the performance of executing a program compiled by the baseline LLVM compiler on an x86-like total-store-order (TSO) hardware. On a set of applications from the PARSEC benchmark suite [5], we find that the overhead of DRFx is 6.49% on average when compared to DRF0.

2. Background

This section provides the motivation for the DRFx memory model and reviews its guarantees and requirements.

2.1 DRF0-based Memory Models

Mainstream programming languages have recently converged on DRF0-based [1] memory models [9, 30]. These models assume that program variables are properly labeled as *synchronization* variables (using annotations such as `volatiles` or `atomic`) and *data* variables. A *data race* occurs when two threads simultaneously access a data variable and at least one of them is a write. A program is said to contain a data race if there exists a sequentially consistent execution in which a data race occurs. A program is *data-race free* otherwise. Programs containing one or more data races are simply referred to as *racy* programs.

DRF0-based memory models guarantee sequential consistency for data-race-free programs while permitting the compiler and the hardware to perform a large class of optimizations on data variables. A programmer still has the flexibility of writing lock-free code and using shared memory for synchronization, provided that appropriate variables are labeled as synchronization in order to satisfy the data-race-free requirement.

2.2 Need for a Fail-stop Semantics for Racy Programs

While DRF0-based memory models provide strong (SC) guarantees for data-race-free programs, they only provide weak guarantees for racy programs. For instance, the C++ memory model treats data races as program errors and provides no semantics for a racy program (akin to programs with buffer-overflow errors). The compiler and the hardware are therefore allowed to introduce arbitrary effects into a racy program, which can compromise the program's correctness and safety. Indeed, it is possible for reasonable compiler optimizations to introduce security vulnerabilities in the presence of data races [9].

Further, while data races are usually considered program errors, they are easy for programmers to inadvertently introduce and difficult to detect. Therefore, when debugging a large C++ system, the programmer has to assume the possibility of a data-race in some component of the system and therefore reason about the interplay of optimizations with races to understand the program's behavior. Type-safe languages like Java mitigate these problems by providing strong-enough guarantees for racy programs to ensure important memory-safety properties [30]. However, this weak semantics is subtle and complex for programmers to reason about, and it has been challenging to ensure that compiler and hardware optimizations in fact respect this semantics [13, 38].

One possible solution to this problem is to design languages and static analyses that simply prevent programmers from introducing data races, and this is an active area of research (e.g., [7, 10, 11]). However, current approaches typically only account for lock-based synchronization, are overly conservative due to imprecise information about pointer aliasing, and require programmer annotations. A promising alternative is to instead provide a fail-stop semantics for racy programs [8, 16, 17, 19, 29, 32]. In this style, the runtime system dynamically detects data races and terminates execution of the program before the non-SC behavior becomes observable to the rest of the system. Therefore, all program executions are guaranteed to behave in an SC manner, despite the possibility of data races.

2.3 Fail-Stop Semantics Requires a Compiler-Hardware Co-design

We argue that efficiently providing fail-stop semantics for racy programs requires an end-to-end solution that involves a co-design of both the compiler and the hardware. Note that such a fail-stop mechanism has stringent requirements on correctness and performance. First, it is unacceptable to halt a data-race-free program. This means that any data-race detection mechanism cannot have false error reports. Second, the overhead of any fail-stop mechanism has to be smaller than the performance obtained from compiler and hardware optimizations. Otherwise, one is better off simply preventing all non-SC transformations in the compiler and the hardware.

The performance requirement clearly rules out existing software-based data-race detection algorithms that add a runtime overhead of 8x or more [18]. More fundamentally, any software-based approach must take care not to introduce data races as part of the detection mechanism itself, since that would expose the mechanism to the weaker guarantees provided by the lower layers of the runtime platform, making it difficult to ensure correctness. All existing data-race detection algorithms maintain some metadata per variable and update the metadata on an access to the variable. When performed without synchronization, these updates can introduce data races in an otherwise data-race-free program.[2] However, as these updates happen at every memory access, adding synchronization for these updates would prevent most optimizations and thereby add significant overhead.

On the other hand, a fail-stop mechanism cannot be implemented in the hardware without any support from the compiler. The weak semantics provided by the Java and C++ memory models makes it impossible to precisely detect source-level data races in the binary. For example, a compiler optimization could introduce data races, causing the hardware to detect a false race. Further, a compiler optimization could *remove* a data race, causing the hardware to fail to detect an actual race. Both the Java and C++ memory models allow the compiler to perform transformations that have these effects [37].

2.4 DRFx Memory Model

Our fail-stop semantics for racy programs is made precise by the DRFx memory model [32], which introduces the notion of a dynamic *memory model (MM) exception* that halts a program's execution. The model ensures the following properties for any program P:

- **DRF:** If P is data-race free then every execution of P is sequentially consistent and does not raise an MM exception.
- **Soundness:** If an execution is not terminated with an MM exception, then that execution is SC.
- **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.

These properties provide a simple and strong guarantee to programmers. Any program execution that does not raise an MM exception is guaranteed to be SC. Therefore, a program execution can always be analyzed and debugged under an assumption of SC behavior, without requiring the programmer to know whether the program has a race. Further, if an execution of P raises an MM exception, then the programmer knows that the program definitely has a data race.

Despite the strong guarantees in the model, the properties are designed to allow flexibility in an implementation. First, a racy program execution need not be halted unless the execution violates SC. This means that one does not require a full-blown data-race detector. On the other hand, the implementation still has the choice to halt a racy execution even if it does not violate SC. This means that one does not require a precise SC-violation detector either. Finally, the Soundness property only ensures that an SC violation will *eventually* be caught, thereby allowing for a lazy detection scheme in the hardware. However, the Safety property requires that violations are at least caught before the next system call, thereby prohibiting undefined program behavior from becoming externally visible.

2.5 The Compiler-Hardware Contract

Our earlier work on DRFx describes a set of requirements for a compiler and hardware implementation that provably ensure the DRFx properties defined above [32]. The key idea is for the compiler to partition a program into single-entry, multiple-exit portions called *regions*. All compiler optimizations are restricted to instructions within a region. In addition, DRFx imposes two restrictions on the compiler. First, any transformation should be *sequentially valid*, meaning that executing the region in isolation from an arbitrary program state should take the program to the same state before and after the transformation. This restriction allows many common optimizations disallowed by the SC memory model, such as common subexpression elimination and register promotion. Second, optimizations are not allowed to introduce reads and writes that are not present in the original program, as they can introduce a data race.[3] However, an optimization is allowed to eliminate reads and writes provided it satisfies the first condition above.

Each synchronization operation must be placed in its own region, thereby preventing reorderings across such accesses. To ensure the DRFx model's Safety property, each system call must also be placed in its own region. Otherwise, region boundaries are unconstrained. The compiler inserts a fence instruction at the entry of each region in order to communicate these boundaries to the hardware.

The hardware must also respect region boundaries (as enforced by memory fences), and it additionally detects conflicting accesses to the same memory location (i.e., accesses where at least one is a write) from concurrently executing regions. If such a region conflict is detected, the hardware throws a memory-model exception and the detected conflict is a witness to a data race in the source program. On the other hand, if there are no region conflicts, then the execution is *region-serializable* — equivalent to an execution in which each region is executed atomically in some global sequential order consistent with the per-thread order of regions — and hence SC.

The following example makes the compiler-hardware contract concrete. Consider the C++ program in Figure 1(a), in which thread

[2] Two threads in a data-race-free program can simultaneously read the same variable. The corresponding metadata updates result in a data race.

[3] DRF0 already precludes a compiler from introducing writes in the program.

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

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

(a)

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

(b)

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

(c)

Figure 1. (a) Program with a data race. (b) Interleaving that exposes the effect of a compiler or hardware reordering. (c) Interleaving that does not. [32]

t uses a boolean flag init to communicate to thread u that variable x has been properly initialized. In the absence of an annotation to declare init a synchronization variable (e.g., volatile in Java [30] or atomic in C++0x [9]), this program is racy. Suppose the compiler puts A and B in a single region, and similarly for C and D. Because A and B are in the same region, they can be potentially reordered, as shown in Figure 1(b). The execution depicted in that figure has a concurrent region conflict and so will be halted with an MM exception. Indeed, if not halted this execution will result in a null pointer dereference, which cannot happen on any SC execution of the original program. On the other hand, the execution shown in Figure 1(c) has no concurrent region conflict and so will not be halted. This execution exhibits SC behavior despite the instruction reordering.

2.6 Realizing the DRFx Compiler and Hardware Design

In this paper we present a complete realization and evaluation of a DRF*x*-compliant platform, including both a conforming compiler adapted from our earlier DRF*x* work and a hardware simulator. Our approach is based on two high-level ideas [32]. First, conflict detection for a region is performed *lazily*, once the region has completed execution. This design supports conflict detection without requiring modifications to the processor's cache architecture or coherence mechanism, which would otherwise be necessary to avoid detecting false conflicts. Second, the compiler bounds the size of each region to ensure that conflict detection can be performed using finite hardware resources. While smaller regions limit the scope of optimizations, a notion of *soft fences* allows the hardware to safely optimize across most region boundaries.

Compared to a traditional memory model based on DRF0 [2], DRF*x* has two main sources of inefficiency that must be assessed. First, there is a loss of performance due to the restriction to only optimize within a region. Second, there is additional overhead due to the region conflict detection mechanism. We previously built a DRF*x*-compliant compiler and used it to evaluate the first cost above [32], but the hardware design was not realized. In this paper, we propose a concrete DRF*x* compliant hardware design, formalize the design along with several important optimizations and prove the design's correctness, implement the design in a hardware simulator, and conduct an end-to-end performance evaluation versus the traditional DRF0 model as well as TSO hardware.

3. DRFx Processor Design

In this section, we discuss our proposed DRF*x* processor architecture. We describe lazy conflict detection using bloom filter signatures, and several optimizations that allow efficient execution in spite of the small, bounded regions created by the DRF*x* compiler. This section discusses the intuition behind the correctness of these optimizations while a more formal presentation will be given in Section 4.

3.1 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 2.5). Figure 3 presents an overview of our DRF*x* hardware design which 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.

Unlike many hardware transactional memory designs, data-race detection requires the hardware to perform conflict detection at byte granularity. Performing precise byte-level eager conflict detection complicates the coherence protocol and cache architecture [29]. For instance, such a scheme would require us 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.

Instead, we employ lazy conflict detection [21]. Each processor core has a *region buffer* which stores the physical addresses of memory accesses executed in a region. An entry is created in the region buffer when a region executes a memory access. A memory access in a region *completes* its execution when it is *committed* from the reorder buffer and, in the case of stores, *retired* from the store buffer. When all the memory accesses in a region have completed their execution, 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 lack of conflicts, it commits the region and reclaims the region buffer entries. We reduce the communication and conflict check overhead by using bloom filter signatures to represent sets of addresses [14]. A *signature buffer* is used to store the read and write signatures for all the in-flight regions in a processor core.

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 our DRF*x* compiler. The static analysis used by our DRF*x* compiler to guarantee this bound is necessarily conservative and may create regions that are much smaller than the desired bound. Frequent soft-fences leads to frequent conflict checks. We reduce this cost by coalescing the soft-fenced regions at runtime. We can support this optimization by using a region buffer slightly larger than the maximum possible region-size guaranteed by the compiler.

To execute a hard fence, a processor has to stall the execution of all future memory accesses until the operations before the hard fence have completed. Forcing the frequent soft-fences introduced by a conservative compiler analysis to adhere to such strict memory ordering requirements results in prohibitive performance overhead due to the hardware being unable to exploit instruction and memory level parallelism across small, soft-fenced regions.

We show that treating a soft fence as a hard fence is unnecessary and indeed leads to significant performance loss. We make several observations that enable the hardware to optimize across soft fences. We show that we can allow memory accesses from a region to execute even if earlier regions that are separated by soft fences have not committed. In addition, we demonstrate that regions separated by a soft fence can be committed out of order. The formal proofs outlined in Section 4 take these optimizations into account and establish that the DRFx runtime requirements are still satisfied.

3.2 Signature-based Lazy Conflict Detection

Let us assume that a processor treats soft fences similar to hard fences, an assumption that we will relax later in the discussion. DRFx 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*. A region buffer entry stores the physical address of a memory access in a region. The DRFx compiler bounds the size of a soft-fenced region to defined bound B, which determines the minimum size that a processor needs to provision for a region buffer.

Similar 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 in the decode stage of the pipeline. Its effective address is eventually written to the region buffer once it is resolved, but before issuing the memory access.

Once all the memory accesses of a region have committed from the re-order buffer (ROB), and stores are retired from the store buffer, the corresponding processor broadcasts the address set to the other processors to perform conflict checks. On receiving a conflict check request, a processor detects a conflict if the addresses in its region buffer intersect with the address set received. If the intersection is empty, an acknowledgment is sent to the requester. On receiving acknowledgments from all the other processors, a processor commits a region by deleting its address entries from the region buffer.

Broadcasting addresses accessed by every region and checking their membership in every processor's region buffer is clearly expensive. To reduce this cost, we propose using bloom filter signatures [14]. Memory addresses accessed by a region are represented using a read and a write signature. Signatures for the in-flight regions are stored in the *signature buffer* (more than one region could be in-flight due to our out-of-order execution optimizations discussed later in Section 3.5). To perform conflict checks for a region, a processor first broadcasts only its signatures. Each processor performs AND operations over the incoming signatures with the contents in its signature buffer. On detecting a potential conflict, a NACK is sent to the requester. On receiving a NACK, a processor sends the full address set for the region so that precise conflict detection can be performed.

The size of signatures needs to be large enough so that false conflicts are rare, avoiding frequent transmission of full address sets. On the other hand, large signatures could incur significant communication overhead. We observed that the average dynamic region size is relatively small (36 in our experiments). But, since we allow many regions to 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, we use large signatures (1024 bits) to avoid false conflicts, but compress the signatures before transmission to reduce communication overhead. Because many regions have small access sets, their signatures are effectively compressed using a simple, efficient run-length encoding scheme (RLE). Using this strategy, we observed very high compression ratios which significantly reduced communication overhead.

Note that our conflict detection architecture does not require additional state to be maintained in the cache, nor does it require changes to the coherence protocol as the DRFx conflict check messages are independent of coherence messages.

3.3 Concurrent Region Conflict Check and Region Execution

We observe that when a processor P receives a conflict check request for 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 address that gets resolved for R during the conflict check can be shown to have executed after the memory accesses in R$'$. Thus, we can order R$'$ before R in the region serialization of the execution.

However, care must be taken to not raise a false conflict over a speculative memory access. The region buffer entry and signature buffer is updated once the address for a memory access is resolved. It is possible that a branch before the memory access is mispredicted, and therefore there is a risk that the memory access could get aborted in future. To avoid raising false exceptions, once a processor detects a conflict, it delays the exception until the conflicting memory access is committed from the ROB. If the memory access gets aborted due to misprediction, then an acknowledgment is sent if there were no other conflicts for the check. In our experiments, we observed that only very rarely we detect a conflict with a memory access in the wrong path. Therefore, the cost of delaying a response to a conflict check due to such conflicts is negligible.

The signature for a region is updated when one of its memory access' address is resolved. When a memory access is aborted due to a branch misprediction we do not update the signatures corresponding to its region. This could result in additional false positives, but we expect the performance impact to be negligible.

3.4 Coalescing Soft-Fence-Bounded Regions

The DRFx compiler uses a conservative static analysis to estimate the maximum number of instructions executed in a region. This could result in frequent soft fences. We observe that dynamically a processor can ignore a soft fence if the preceding soft-fenced region executed fewer memory accesses than a pre-determined threshold T. Combining two contiguous soft-fenced regions at runtime does not violate DRFx 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. Our 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, because the probability of aliases in signatures increase. Also, it could undermine out-of-order commit optimization.

3.5 Out-of-Order Execution and Commit of Regions

In the discussion so far, we have assumed that a soft fence behaves like a hard fence. However, we observe that two important restrictions that need to be obeyed for hard fences can be relaxed for soft fences, which allows us to attain performance close to DRF0.

First, we allow out-of-order execution of soft-fenced regions. In the case of a hard fence, before a processor can execute memory accesses from a region, it has to wait for all the memory accesses in the preceding regions to complete. We refer to this restriction as *in-order execution of regions*. This is clearly a requirement for hard fences, since we may detect false conflicts if memory accesses are allowed to be reordered across hard fences that demarcate synchronization operations. However, we observe that this memory ordering can be relaxed for soft fences. For the example in Figure 2, 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 (I_1) in previous regions, its value can be forwarded to a load in a later region (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 needs to be conflict checked. Therefore, it does not mask any conflicts that would have been detected before. Also, aggressive reordering of soft-fenced regions does not reorder accesses across synchronization operations, so any conflicts detected guarantees presence of data races.

Second, we observe that once a region's memory accesses are completed, a processor can initiate conflict checks and commit the region from the region buffer if the check succeeds. Since the ROB commits instructions in-order, it is guaranteed that when a region is ready to commit, all the memory accesses from 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. Under this scenario, we claim that it is correct to conflict check and commit a later region as long as all its accesses have committed from the ROB and retired from the store buffer.

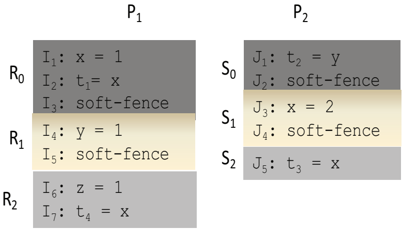

Figure 2. An Example Binary Compiled Using DRFx Compiler.

For example, in Figure 2, 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. We claim that a processor can perform conflict checking for R_1 and commit by deleting its corresponding entries from the region and signature buffers. This optimization can be intuitively understood by observing that even if R_1 commits, as long as R_0 does not raise a conflict, there would be a global serial order for all the soft-fenced regions.

These relaxations on concurrent conflict detection and region execution obviate the need for the arbiter used in other lazy conflict detection schemes [14, 21]. The formalism presented in Section 4 accounts for out-of-order region execution and commit.

3.6 Handling Context Switches

A thread can incur a context switch at runtime for a variety of reasons. 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 adversarial programs that can perform unbounded computation (while still performing a bounded number of memory accesses), we require that the DRFx 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 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 DRFx-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. This observation leads to a simple solution that does not require virtualizing the region buffer.

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 8 context switches.

3.7 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. Exact addresses of the conflicting instructions could be provided with additional storage overhead by extending each region buffer entry to track the program counter (PC) of each memory instruction.

A processor may encounter non-MM exceptions such as a null-pointer dereference, division by zero, etc., 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, it raises the non-MM exception. But if a conflict is detected, 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 [29]. The state could still be non-SC with respect to the source program due to compiler optimizations.

3.8 System Calls and Safety

The DRFx 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 DRFx compiler requirement that every region's size be bounded to a predefined limit. When a program executes a region that exceeds the bound, the

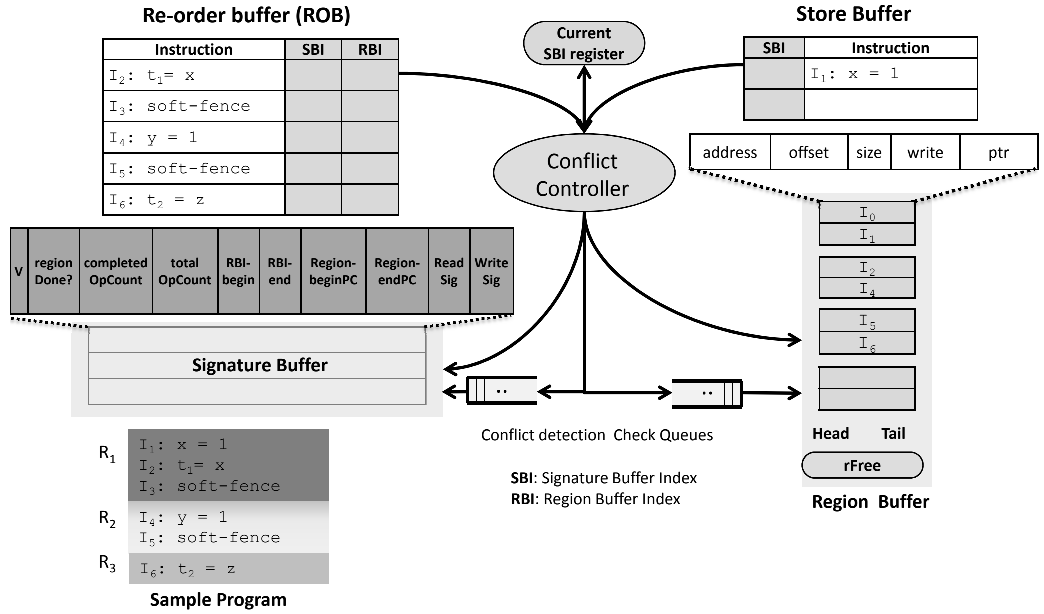

Figure 3. Architecture Support for DRFx (shown in gray).

DRFx hardware can trivially detect that condition and raise an MM exception to ensure safety.

3.9 DRFx 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 which we extend with additional messages to support conflict checking. Conflict check messages are independent of coherence messages. Figure 3 shows our DRFx hardware extensions to a baseline out-of-order processor with store buffer. We now describe these extensions in detail.

When a hard fence is decoded, 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 fetched and decoded. When a soft fence is decoded, we create a new region only if the current region size (stored in the `totalOpCount` field of the region's signature buffer entry) exceeds a pre-determined threshold `T` (32 in our experiments). If a soft fence does not start a new region, it is turned into a `nop` in the decode stage. If a hard or soft fence starts a new region, 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.

When a memory instruction is decoded, a region buffer entry is allocated for it. The register `rFree` keeps track of the total number of free entries in the region buffer. If no free region entry is available, the memory access is stalled in the decode stage of the pipeline. Since regions can commit out-of-order, we maintain a linked-list to track the free region buffer entries. The `head` and `tail` registers point to the first and last entries in the free list respectively. Free entries, and also those allocated to a region are linked through the `ptr` fields. The `ptr` for the first memory access of a region is set to null. The `ptr` for any later memory access of a region points to its immediately preceding memory access's region buffer entry. 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).

Instead of allocating one region buffer entry for each memory access in a region, we could potentially coalesce accesses to the same memory location into one entry. We opted not to do this in order to simplify the design, such as the logic required for deleting region buffer entries when memory accesses get aborted due to a branch misprediction.

The number of memory accesses decoded as part of the current region is maintained by incrementing the `totalOpCount` field in the current region's signature buffer entry (specified by the `SBI` register). Completion of memory accesses of a region is maintained by incrementing the `completedOpCount` field when a load commits from the ROB or a store retires from the store buffer. These counts are used to determine when a region is ready to commit.

When a memory operation's address is resolved, completed or aborted, corresponding entries in the signature and region buffer need to be updated. To determine these entries, each ROB entry contains two pointers: the Region Buffer Index (RBI) that points to the memory access' region buffer entry, and the SBI of the region that the memory access corresponds to.

When a memory access is aborted due to a branch misprediction, its region buffer entry is deallocated and its region's `totalOpCount` is decremented. Note that a store in the store buffer does not need the RBI pointer as it is non-speculative. It does however need the SBI pointer to update the `completedOpCount` in the signature buffer when it retires.

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 `completedOpCount` is equal to `totalOpCount`. 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, the region is committed by deallocating its entries in the signature and region buffers. The signature buffer entry is identified using the region's SBI used during its 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. Committed region's region buffer entries are added to the free list.

3.9.1 Region Buffer and Signature Buffer Sizes

The number of entries in the region buffer needs to be at least as large as the bound assumed by the DRFx compiler for a soft-fenced region (512 entries for our study). But a larger buffer allows us to coalesce regions. Also, a larger region buffer could expose additional opportunities for out-of-order execution across regions. We assume a 544-entry region buffer in our evaluation (512 is the compiler bound, additional 32 entries support coalescing).

DRFx requires byte-level, precise conflict detection, but storing an entry for each byte accessed would be inefficient. To minimize region buffer space, we optimize for the common case, which is word-level access. Thus, a region buffer entry includes the word address, byte offset, and the number of bytes accessed. This requires 81 bits per region buffer entry: 8 bytes for double word aligned address, 3 bits for byte offset, 3 bits for access size, a read/write bit, and 10 bits for `ptr`. Thus, the size of a region buffer in each core is 5.38KB.

The region buffer is designed as a banked content addressable memory (CAM). This allows us to perform fast membership tests using word addresses when performing precise conflict checking for a remote region. Only when a potential word-level conflict is detected do we perform a precise byte-level address comparison. We expect that the CAM would be accessed relatively infrequently, only when a signature-based conflict check finds a potential conflict.

Each core has a signature buffer with 17 entries, as there can only be a maximum of 17 in-flight regions in our design with 544-entry region buffer and a coalescing threshold (`T`) of 32. Each entry has a read and a write signature of size 1024 bits to capture the addresses accessed in a region. In addition, each entry contains a valid bit, a `regionDone` bit, `RBI-begin`, `RBI-end`, `completedOpCount`, `totalOpCount` (10 bits each since there are 544 region buffer entries which is also the maximum number of memory operations in a coalesced region), and starting and ending program counter values of a region (64 bits each). Thus, a signature buffer with 17 entries is of size 4.60KB.

4. Correctness of Design

In this section we describe the formal arguments we use to show that our optimized hardware design satisfies the two key requirements for a DRFx-compliant execution environment: exception-free executions are region serializable and only compiled programs that contain a data race can raise an MM exception. Full proofs of the lemmas and theorems are omitted, but can be found in the accompanying technical report [40].

In showing the correctness of the design, we must model certain orderings that hold between various events in the system. Keep in mind that none of the orderings we describe are violated by the optimizations described in Section 3. In particular we make no assumptions about the order in which memory accesses from adjacent soft-fenced regions complete or the order in which those regions commit. We consider three events in a (memory access) instruction's execution: address resolution, commit from the reorder buffer, and (for store instructions) retirement from the store buffer. For any particular instruction, a processor performs each of these events in this order. Any events that can be shown to have a causal relationship within a processor give rise to a partial order on events that is consistent with physical time. We will refer to this partial order as the processor happens before order (PHB). So, for an instruction M, we have $M\ \texttt{resolved} <_{\text{PHB}} M\ \texttt{committed} <_{\text{PHB}} M\ \texttt{retired}$. In an attempt to treat loads and stores more uniformly in the formalism, we will also refer to an instruction being *completed*. We consider a store completed when it has retired from the store buffer (M a store and M $\texttt{retired} \Rightarrow M\ \texttt{completed}$). Loads and all other instructions will be considered completed as soon as they have been committed from the reorder buffer (M not a store and $M\ \texttt{committed} \Rightarrow M\ \texttt{completed}$).

$\langle R \rangle$	Send the conflict detection msg for region R
$\langle P \leftarrow R \rangle$	Receive a conflict detection msg for region R at processor P
$\langle P \checkmark R \rangle$	Send an ack msg for R from P back to originating processor
R `committed`	Commit region R, clearing all its entries in region buffer

Table 1. Events in Conflict Detection

We also include in PHB the causal order between instruction execution events and the events performed as part of conflict detection, which are shown in Table 1. A processor waits for all instructions in a region to be completed (all instructions committed and all stores flushed from store buffers) before sending a conflict detection message. So, for all instructions M in region R we have $M\ \texttt{completed} <_{\text{PHB}} \langle R \rangle$. Furthermore, although regions may be committed out of order, because instructions are committed from the reorder buffer in-order, we know that for all instructions M in regions program-ordered before region R, we have $M\ \texttt{committed} <_{\text{PHB}} \langle R \rangle$. We also know that a processor must receive a request for conflict detection before it performs detection and eventually sends an acknowledgment. We also use PHB to capture this: $\langle P \leftarrow R \rangle <_{\text{PHB}} \langle P \checkmark R \rangle$.

The fact that the conflict detection mechanism does not find a conflict can also indicate events that are ordered by PHB. It is clear that if a processor P is performing conflict detection in response to a request from processor P' for a region R' and there is an address-resolved, conflicting entry in the region buffer for the entirety of the conflict detection process, then an exception will be raised. Furthermore, conflict detection for R' occurs strictly between the time that the request is received and the acknowledgment is sent. This reasoning makes no assumptions about the order of execution of instructions in different regions, the number of regions in flight, the order of region commit, or whether or not conflict detection for multiple requests (or a coalesced request) is proceeding in parallel.

So, the following proposition captures what orderings we can infer if a conflict is *not* detected in our fully optimized design.

Proposition 1. *If M and M' are conflicting memory accesses contained respectively in region R executing on processor P and region R' executing on processor P' where $P \neq P'$, and if the execution does not raise an MM exception, then either:*

1. $R\ \textit{committed} <_{\text{PHB}} \langle P \checkmark R' \rangle$
   or
2. $\langle P \leftarrow R' \rangle <_{\text{PHB}} M\ \textit{resolved}$.

We model the fact that a message must be sent by one processor before it is received by another processor by a partial order called message happens before (MHB). So we have $\langle R \rangle <_{\text{MHB}} \langle P \leftarrow R \rangle$, and also $\langle P \checkmark R \rangle <_{\text{MHB}} R\ \texttt{committed}$ since the originating processor must receive acknowledgments from all other processors before it commits the region. (We've left the reception of the acknowledgment as an implicit event since it won't be needed in the proofs). Since both PHB and MHB are partial orders that arise from causal relationships and are consistent with physical time, we know that their union is acyclic. We will call this partial order the system happens before relation: $\text{SHB} = \text{PHB} \cup \text{MHB}$.

Notice that Proposition 1 is symmetric in M and M'. But, it cannot be the case that both $R\ \texttt{committed} <_{\text{PHB}} \langle P \checkmark R' \rangle$ and $R'\ \texttt{committed} <_{\text{PHB}} \langle P' \checkmark R \rangle$. Otherwise we would have a cycle in $\text{SHB} : R\ \texttt{committed} <_{\text{PHB}} \langle P \checkmark R' \rangle <_{\text{MHB}} R'\ \texttt{committed} <_{\text{PHB}} \langle P' \checkmark R \rangle <_{\text{MHB}} R\ \texttt{committed}$. Neither can it be the case that both $\langle P \leftarrow R' \rangle <_{\text{PHB}} M$ $\texttt{resolved}$ and $\langle P' \leftarrow R \rangle <_{\text{PHB}} M'$ $\texttt{resolved}$. Again, this would contradict the acyclicity of SHB since $\langle P \leftarrow R' \rangle <_{\text{PHB}} M$ $\texttt{resolved}$ $<_{\text{PHB}} \langle R \rangle <_{\text{MHB}} \langle P' \leftarrow R \rangle <_{\text{PHB}} M'$ $\texttt{resolved}$ $<_{\text{PHB}} \langle R' \rangle <_{\text{MHB}} \langle P \leftarrow R' \rangle$. So we incorporate this information to state a stronger version of the proposition.

Proposition 2. *If M and M' are conflicting memory accesses contained respectively in region R executing on processor P and region R' executing on processor P' where $P \neq P'$, and if the execution does not raise an MM exception, then either:*

1. *R $\texttt{committed}$ $<_{\text{PHB}} \langle P \checkmark R' \rangle$ and*
   $\langle P' \leftarrow R \rangle <_{\text{PHB}} M'$ $\texttt{resolved}$
   or
2. *$\langle P \leftarrow R' \rangle <_{\text{PHB}} M$ $\texttt{resolved}$ and*
   R' $\texttt{committed}$ $<_{\text{PHB}} \langle P' \checkmark R \rangle$.

We now describe some orderings on memory operations. Our cache coherence protocol guarantees us a total order on writes to the same memory location. We will call this order CO. This can be extended to a partial order EO on loads and stores to the same memory location by ordering a load after the store that it reads and before the store that follows that store in CO. In order to define region serializability, we consider the lifting of EO to regions, denoted by EO^R. We define $R_1 <_{\text{EO}^R} R_2$ to hold if there exist conflicting memory operations $M_1 \in R_1$ and $M_2 \in R_2$ such that $M_1 <_{\text{EO}} M_2$. We can now state the definition of region serializability in terms of EO^R and the program order of regions within a thread, which we will call TO.

Definition 1. *Let the region ordering relation* RO *be defined as* $\text{RO} = \text{EO}^R \cup \text{TO}$. *An execution is* region serializable *if* RO *is a partial order.*

In order to prove that an exception-free execution is region serializable, we first establish the following two lemmas.

Lemma 1. *If a memory access M in region R executing on processor P conflicts with a memory access M' in region R' executing on P' where $P \neq P'$, and if $\langle P' \leftarrow R \rangle <_{\text{PHB}} M'$ $\texttt{resolved}$ and the execution is MM-exception-free, then $M <_{\text{EO}} M'$.*

Lemma 2. *If M and M' are conflicting memory accesses contained respectively in region R executing on processor P and region R' executing on processor P' where $P \neq P'$, and if the execution does not raise an MM exception, and if $M <_{\text{EO}} M'$, then R $\texttt{committed}$ $<_{\text{PHB}} \langle P \checkmark R' \rangle$ and $\langle P' \leftarrow R \rangle <_{\text{PHB}} M'$ $\texttt{resolved}$.*

We use these lemmas to establish the following theorem.

Theorem 1. *An execution on our optimized hardware that does not raise an MM exception is region serializable.*

We further demonstrate that if two conflicting memory accesses are ordered by synchronization operations ($M <_{\text{TO}} S <_{\text{EO}} S' <_{\text{TO}} M'$), then they cannot cause a memory model exception to be raised. This is the result of the stringent ordering constraints that the hardware places on the hard fences surrounding synchronization operations.

Theorem 2. *If a compiled program is free of data races, then our optimized hardware will not raise an MM exception during its execution.*

Full proofs for these theorems and lemmas can be found in [40]. Theorems 1 and 2 can be combined with the results shown for a DRFx-compliant compiler in [31] in order to establish the end-to-end DRFx guarantees.

5. Results

This section discusses results analyzing the cost of supporting DRFx, which includes the cost of optimizations lost due to compiler constructed soft-fences and runtime conflict detection.

5.1 Simulation Methodology

We implemented three compiler versions using LLVM[28] to study the performance of TSO, DRF0 and DRFx. The TSO compiler is the baseline LLVM compiler at `-O3` optimization level. The DRF0 compiler inserts a hard fence using the `llvm.memory.barrier` intrinsic before and after every synchronization call and access to a volatile variable. Also, we disabled speculative optimizations. [4] For the DRFx compiler, we inserted soft fences before performing any compiler optimization to bound region sizes to 512 memory accesses. The size is estimated using a conservative static analysis over the control-flow-graph. In addition, our compiler conservatively adds a soft fence at every back-edge of a loop.

We modeled three hardware architectures, TSO, DRF0 and DRFx using a cycle-accurate, execution driven, Simics based x86_64 simulator called FeS2 [34]. Our processor configuration is shown in Table 2. The model includes a MOESI coherence protocol and a store buffer to hold pending stores that have been committed from the ROB. For TSO, the store buffer is organized as a FIFO queue to enforce in-order retirement of stores. For DRF0 and DRFx, the store buffer can retire stores between two hard fences out-of-order. We also allow two stores to the same cache block to be coalesced into one entry in the store buffer for the DRF0 and DRFx models.

We modeled a hierarchical switch network with a fan-out degree of 4, 512-bit link width, 1-cycle link latency. We assume a virtual channel (VC) for each message type: address, control, broadcasting signatures in DRFx, and unicast messages in DRFx (request full address set, reply full address set, acknowledgments). We also modeled 2-cycle latency for issuing a message into the interconnect. Limited message buffer capacity at each switch was not modeled.

Our model includes speculative load execution support for TSO [20]. It allows a load to be speculatively reordered before another load. A misspeculation is detected when a processor receives an invalidation request for a cache block that was read speculatively by a load before that load commits from the ROB. In the event of a misspeculated load, the load and its subsequent instructions are squashed and re-executed. This optimization is not necessary for DRF0 and DRFx as they can re-order loads and stores between two hard fences.

For all the models, memory accesses are not allowed to execute until preceding hard fence has committed. A hard fence is committed only after all the preceding memory accesses have retired from the store buffer.

For DRFx we modeled a region buffer of size 512 (compiler bound) + 32 (to support region coalescing). This buffer is divided into 8 banks and each bank is a CAM. We assume 2 cycle latency for associative accesses to this structure, which we estimated using CACTI [12]. For signature based designs, we use a 1024 bit signature and use the hash used in Bulk [14]. We used a signature buffer of size 17, which allows a maximum of 17 regions to be in-flight in a processor core.

We evaluated the performance of various designs over PARSEC [5] and SPLASH-2 [43] benchmarks. Table 3 lists the IPC for the baseline DRF0 model. All of these benchmarks are run to the completion. For PARSEC benchmarks, we used the `sim-medium`

[4] Specifically, we modified `llvm::isSafeToLoadUnconditionally`, `Instruction::isSafeToSpeculativelyExecute` and `MachineInstr::isSafeToMove` to return false if instruction is a load

input set (except for streamcluster, for which we used the `sim-small` input). For SPLASH-2 applications we use the default inputs.

Processor	4-core CMP. Each core operating at 2Ghz.
Fetch/Exec/Commit width	4 instructions (maximum 2 loads or 1 store) per cycle in each core.
Store Buffer	TSO: 64 entry FIFO buffer with 8 byte granularity. DRF0, DRFx: 8 entry unordered coalescing buffer with 64 byte granularity.
L1 Cache	64 KB per-core (private), 4-way set associative, 64B block size, 1-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, 1-cycle link latency.
Memory	80-cycle DRAM lookup latency.
RegionBuffer	544 entry, 8 banks, 2-cycle CAM access.
Bloom filter	1024 bits. 2 banks indexed by 9 bit field after address permutation[14]. 2-cycle access latency.

Table 2. Processor Configuration

Application	Avg. IPC (DRF0)	Application	Avg. IPC (DRF0)
blackscholes	1.97	bodytrack	1.75
canneal	0.27	facesim	0.53
streamcluster	1.61	barnes	1.59
fft	1.41	radix	0.99
raytrace	0.59		

Table 3. Average IPC for DRF0

5.2 DRFx Memory Model Performance Comparison

Figure 4 compares the performance of TSO, DRF0 and DRFx. The results are normalized to the execution time of DRF0. For DRFx, we show two results. The first result labeled as `DRF0 + soft-fence` represents the compiler cost of DRFx and is the maximum performance our DRFx hardware can hope to achieve. To obtain this result, we executed a program compiled using DRFx compiler on DRF0 hardware, treating every soft-fence as a no-op. The second result for DRFx is for a processor configuration that employs all the optimizations we discussed in this paper. It represents the compiler and hardware conflict detection cost to support DRFx.

We find that the average performance overhead to support DRFx is about 6.49% when compared to DRF0. `barnes` has the highest overhead of 24%. DRFx conflict detection adds only about 2.3% higher overhead to `DRF0 + soft-fence`. As demonstrated in the next section, our optimizations are crucial for achieving close to the best possible hardware performance.

The remaining cost in DRFx is due to restricting compiler optimizations to soft-fence bounded regions. For example, DRFx model for `bodytrack` is about 12.84% slower than DRF0, but almost all of this overhead is due to the DRFx compiler. Our current compiler bounding analysis is very conservative, especially regarding loops, and so we believe that this cost could be significantly improved.

We found that soft fences constitute 6% of total committed instructions. Average dynamic size of compiler constructed soft-fenced regions is about 10 memory operations. However, our runtime coalescing optimization increases this to 36 memory operations. We could improve this further by increasing the region buffer size. Small regions result in frequent conflict checks, but compressing signatures avoids excessive communication overhead for these checks. We find that run-length encoding of signatures compresses signatures by 4.34x on average, significantly reducing network bandwidth requirement. The amount of data communicated processor cores increased by 84% due to conflict messages in DRFx when compared to DRF0. More aggressive coalescing could potentially reduce this overhead further.

TSO represents the performance gotten by compiling a program using a stock compiler and executing it on stock hardware. We found that TSO is nearly as fast as DRF0 on our benchmarks.

5.3 Effectiveness of Out-of-Order Optimizations

Figure 5 compares the effectiveness of three of our optimizations: out-of-order region execution, out-of-order region commit, and coalescing contiguous regions. All of the configurations assume signature-based conflict detection. The performance results are normalized to the DRF0 execution time.

If we had not distinguished between hard and soft fences, none of the above three optimizations would have been feasible. This configuration is represented as `io-exec, io-commit`. We find that the performance overhead without the three optimizations would be on average 121% slower than DRF0. Allowing regions to execute out-of-order brings the overhead down to about 27%. Allowing regions to commit out-of-order further reduces the overhead down to about 7.7% on average. Finally, coalescing regions at runtime reduces the overhead to about 6.49%. This optimization is especially effective for programs for which our conservative compiler bounding analysis constructs very small regions. For instance, on `streamcluster` the overhead is reduced from 36% to about 15%.

Conflict detection in our design can negatively impact performance by stalling the execution of a processor core only under two circumstances. One, a region buffer entry is not available for a memory operation to issue. Two, even after all the memory accesses in the regions preceding a hard fence have completed, the processor must stall to allow conflict detection to complete. We measured the proportion of the processor cycles that are stalled due to these two reasons for a configuration with in-order commit and one allowing out-of-order commit. The results are shown in Figure 6. We observe that out-of-order commit significantly reduces both kinds of stalls for most benchmarks.

5.4 Effectiveness of Signature-Based Conflict Detection

Figure 7 compares the performance of DRFx with and without signature based conflict check optimization. We observe that, on average, the signature based scheme improves performance by 7%. The average false positive rate for bloom filter checks was 1.39% and the maximum was 12% on `bodytrack`.

5.5 Scalability

Figure 8 shows the performance of DRFx when compared to the baseline DRF0 as the number of cores scale. We find the network bandwidth is adequate to provide scalable performance up to 16-cores. For `blackscholes` there is a noticeable performance difference as we go from 4 to 8 cores. This is due to an increased number of false conflicts when signatures are compared to a larger number of regions on more cores. On detecting a potential conflict, we have to perform expensive, precise conflict check over the complete set of addresses. For `blackscholes`, precise conflict detection was performed for about 13% of regions in the 8-core configuration as opposed to only 5% in the 4-core configuration.

5.6 Exceptions

We detected conflicts due to a few benign data races in the applications we studied. One source of benign races was programmer constructed barrier synchronizations and semaphores. These synchronization variables were not typed as `volatile` in the program which caused the conflicts. Also, we detected two conflicts due to benign data races in `glibc` – one in `__drand48_iterate()` and another in `__cfree` and `malloc_consolidate` functions. To ensure correctness even under DRF0, a programmer needs to correctly flag the racy variables in these functions as `volatile`.

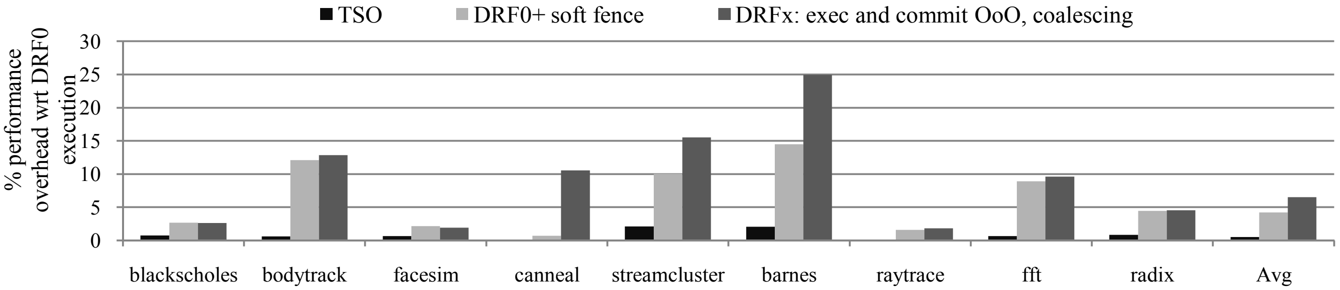

Figure 4. Performance of the three memory models we evaluated. `DRF0 + soft-fence` represents only the compiler cost of DRFx

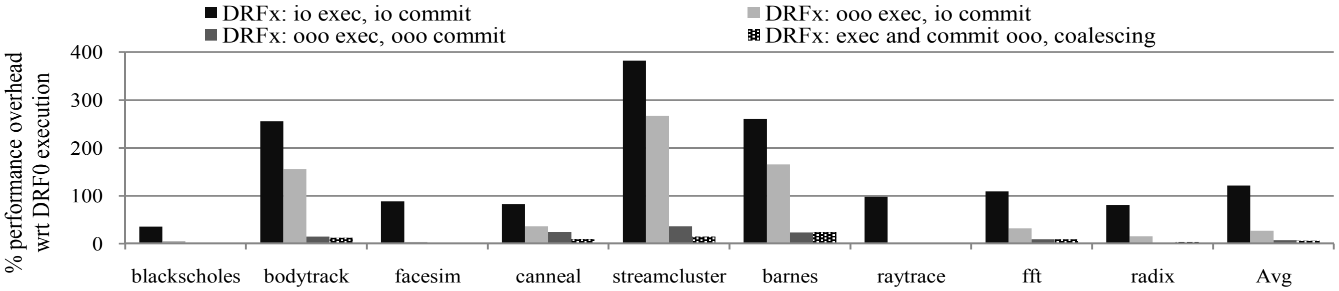

Figure 5. Effectiveness of Region Coalescing, and Out-Of-Order Region Execution and Commit Optimizations.

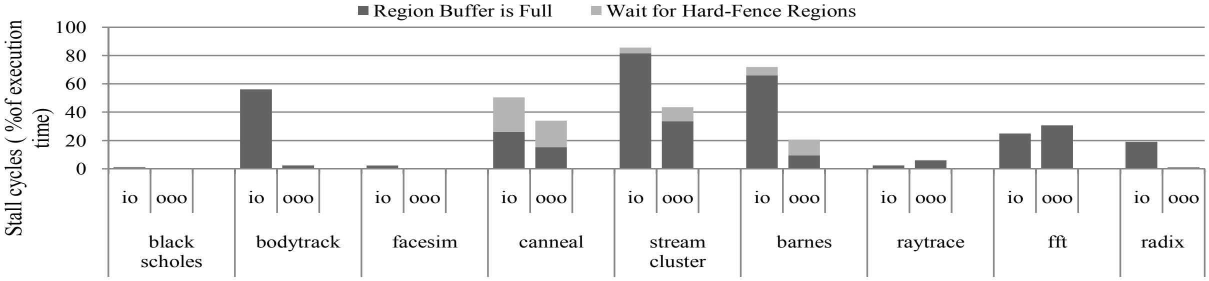

Figure 6. Processor Stall Cycles Due to Conflict Detection

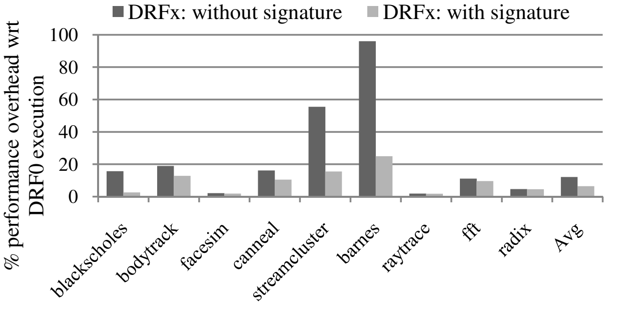

Figure 7. Effectiveness of Signature Based Conflict Detection

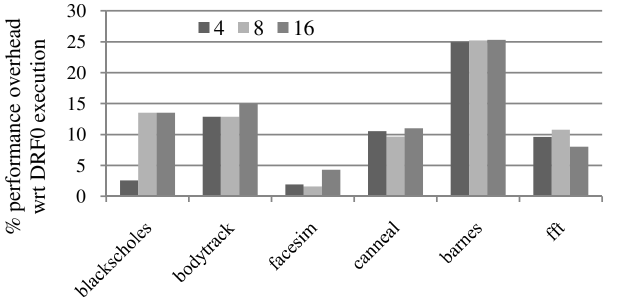

Figure 8. Performance overhead of DRFx for 4, 8 and 16 cores with respect to the DRF0 performance for the same number of cores.

6. Related Work

This section discusses the most closely related work.

6.1 Memory Models With Exceptions

The C++ memory model [9] and the Java memory model [30] are based on DRF0 [1] and share its limitations for racy programs which we discussed in Section 1.

Concurrently with our earlier work on DRFx, Lucia *et al.* defined *conflict exceptions* [29], which also use a notion of regions to detect language-level SC violations in hardware. Their approach can be viewed as a realization of DRFx-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. [3] proposed to detect data races at runtime using hardware support. Elmas et al. [17] augment the Java virtual machine to dynamically detect bytecode-level data races and raise a `DataRaceException`. Boehm [8] 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 8x or more performance overhead in software [18] or incurs significant hardware complexity [33, 35]. A full data-race detector is inherently complex as it has to dynamically build the *happens-before* graph [26] 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, etc. In contrast, DRFx hardware is inherently simpler as it requires that we track memory access state and perform conflict detection over only the uncommitted, bounded regions.

Gharachorloo and Gibbons [19] 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. DRFx 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.

6.2 Efficiently Supporting Sequential Consistency

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

Bulk compiler [4] together with the BulkSC hardware [15] 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.

DRFx hardware could be simpler than Bulk hardware as it avoids the need for speculation (especially across I/O) and unbounded region sizes which have been the two main issues in realizing a practical transactional memory system. However, DRFx 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 [24, 39, 41]. However, computing a minimal set of fences for a program is NP-complete [25]. One approach to reduce the number of fences is to statically determine potentially racy memory accesses [24, 41] 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 [10, 11], 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 [6, 15, 36, 42].

6.3 Transactional Memory

Hardware transactional memory (HTM) systems [23] also employ conflict detection between concurrent regions. However, unlike TM systems, regions in DRFx 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 DRFx system does not suffer from the two issues that have been most problematic for practical adoption of TM.

Hammond et al. [22] proposed transactional coherency and consistency (TCC) memory model based on a transactional programming model [23]. 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, DRFx 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. [22] 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, DRFx 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.

7. Conclusion

The DRFx memory model provides strong and easy-to-understand guarantees for both data-race-free and racy programs while safely supporting most sequentially valid compiler and hardware optimizations. A significant challenge in bringing DRFx to practice is the cost of runtime conflict detection. Processor support can reduce this cost, but it can be realized in practice only if it is both efficient and simple.

This paper addresses this challenge through careful design choices and optimizations of a DRFx-compliant micro-architecture. We employ a lazy conflict detection design in order to avoid modifying the existing cache architecture and coherence mechanism. We leverage interaction with the compiler to partition a program into bounded-size regions, thereby avoiding the need to handle overflow of hardware resources during conflict detection. Further, the cost of bounded-size regions is significantly reduced by several novel optimizations that we have formalized and proven correct.

We used a DRFx-compliant compiler and a hardware simulator to evaluate our design. We find that the average performance overhead is about 6.5% when compared to DRF0 memory model.

Acknowledgments

We thank our shepherd Kunle Olukotun for comments that helped us improve this paper. 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.

References

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

[2] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *ISCA '90*, pages 2–14. ACM, 1990.

[3] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA '91*, 1991.

[4] W. Ahn, S. Qi, J.-W. Lee, M. Nicolaides, X. Fang, J. Torrellas, D. Wong, and S. Midkiff. BulkCompiler: High-performance sequential consistency through cooperative compiler and hardware support. In *42nd International Symposium on Microarchitecture*, 2009.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of PACT*, October 2008.

[6] C. Blundell, M. Martin, and T. Wenisch. InvisiFence: performance-transparent memory ordering in conventional multiprocessors. In *ISCA '09*, pages 233–244, 2009.

[7] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.

[8] H. J. Boehm. Simple thread semantics require race detection. In *FIT session at PLDI*, 2009.

[9] H. J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08*, pages 68–78. ACM, 2008.

[10] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01*, pages 56–69, 2001.

[11] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA '02*, pages 211–230, 2002.

[12] Cacti. Hp labs. cacti 4.2. URL `http://quid.hpl.hp.com:9081/cacti`.

[13] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP '07*, 2007.

[14] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06*, 2006.

[15] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA '07*, pages 278–289, 2007.

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

[17] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *PLDI '07*, pages 149–158, 2007.

[18] C. Flanagan and S. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09*, pages 121–133, 2009.

[19] K. Gharachorloo and P. Gibbons. Detecting violations of sequential consistency. In *SPAA '91*, pages 316–326, 1991.

[20] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of ICPP*, volume 1, pages 355–364, 1991.

[21] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI*, pages 1–13, 2004.

[22] L. Hammond, V. Wong, M. K. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, pages 102–113, 2004.

[23] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93*, pages 289–300. ACM, 1993.

[24] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in Titanium. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 15. IEEE Computer Society, 2005.

[25] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38(2):130–144, 1996.

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

[27] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 100(28):690–691, 1979.

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of CGO*. IEEE Computer Society, 2004.

[29] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm. Conflict exceptions: Providing simple parallel language semantics with precise hardware exceptions. In *ISCA '10*, pages 210–221, 2010.

[30] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *POPL '05*, pages 378–391. ACM, 2005.

[31] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A simple and efficient memory model for concurrent programming languages. Technical Report 090021, UCLA Computer Science Department, Nov. 2009. URL `http://fmdb.cs.ucla.edu/Treports/090021.pdf`.

[32] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A simple and efficient memory model for concurrent programming languages. In *PLDI '10*, 2010.

[33] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. SigRace: signature-based data race detection. In *ISCA '09*, pages 337–348, 2009.

[34] N. Neelakantam, C. Blundell, J. Devietti, M. M. K. Martin, and C. Zilles. FeS2: A full-system execution-driven simulator for x86. In *Poster session at ASPLOS '08*, 2008. URL `http://fes2.cs.uiuc.edu/acknowledgements.html`.

[35] M. Prvulovic and J. Torrelas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of ISCA*, San Diego, CA, June 2003.

[36] P. Ranganathan, V. Pai, and S. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *SPAA '97*, pages 199–210, 1997.

[37] J. Sevcík. Private communication.

[38] J. Sevcík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP '08*, pages 27–51, 2008.

[39] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):282–312, 1988.

[40] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient processor support for DRFx: Technical report. Technical Report 110002, UCLA Computer Science Department, Mar. 2011.

[41] Z. Sura, X. Fang, C. Wong, S. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of PPoPP*, pages 2–13, 2005.

[42] T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *ISCA'07*, 2007.

[43] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA'95*, pages 24–36, 1995.