DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages

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

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

Abstract

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 thereby impacting performance. Data-race-free (DRF) models, such as the proposed C++0x 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 DRFx memory model, which is simple for programmers to understand and use while still supporting many common optimizations. We introduce a *memory model (MM) exception* which can be signaled to halt execution. If a program executes without throwing this exception, then DRFx guarantees that the execution is SC. If a program throws an MM exception during an execution, then DRFx 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 about this model, describe a compiler and hardware design suitable for DRFx, and evaluate the performance overhead due to our compiler and hardware requirements.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages

General Terms Design, Languages

Keywords memory models, sequential consistency, data races, memory model exception, soft fences

1. Introduction

A memory consistency model (or simply *memory model*) forms the foundation of shared-memory multi-threaded programming. It defines the set of possible orders in which memory operations can execute and the possible values a read can return, thereby providing a contract that programmers can assume and that compilers and hardware must obey. While it is desirable to provide programmers with a simple and strong guarantee about the behavior of their programs, doing so can reduce the flexibility of compilers and hardware to perform optimizations and thereby negatively impact performance.

A case in point is the memory model known as *sequential consistency* (SC) [23], which requires all memory operations in an execution of a program to appear to have executed in a global sequential order consistent with the per-thread program order. This memory model is arguably the most simple for programmers, since it matches the intuition of a concurrent program's behavior as a set of possible thread interleavings. However, many program transformations that are *sequentially valid* (i.e., correct when considered on an individual thread in isolation) can potentially violate SC in the presence of multiple threads. For example, reordering two accesses to different memory locations in a thread can violate SC since another thread could "view" this reordering via concurrent accesses to those locations. As a result, SC precludes the use of common compiler optimizations (code motion, loop transformations, etc.) and hardware optimizations (out-of-order execution, store buffers, lockup-free caches, etc.).

In recent years, there have been significant efforts to bring together language, compiler and hardware designers to standardize memory models for mainstream programming languages. The consensus has been around memory models based on the data-race-free-0 (DRF0) model [1], which attempts to strike a middle ground between simplicity for programmers and flexibility for compilers and hardware. In the DRF0 model, a programmer explicitly distinguishes synchronization accesses from other data accesses (using type qualifiers such as `volatile` in Java [28] and `atomic` in C++ [7].) The compiler and hardware are limited in the optimizations and reorderings they can perform across synchronization accesses, in order to ensure their semantics is properly respected. The DRF0 model then guarantees SC for all properly synchronized programs (i.e., programs that are free of *data races*). Unlike the full SC model, DRF0 can achieve good performance since many standard sequentially valid compiler optimizations preserve SC for properly synchronized programs.

The DRF0 model provides a simple and strong guarantee for race-free programs, but it does not specify any semantics for programs that contain data races. While such programs are typically considered erroneous, data races are easy for programmers to accidentally introduce and are difficult to detect. The DRF0 model therefore poses two important problems for programmers:

- Since a racy execution can behave arbitrarily in DRF0, it can violate desired safety properties. For example, Boehm and Adve

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

show how a sequentially valid compiler optimization can cause a program to jump to arbitrary code in the presence of a data race [7].

- Debugging an erroneous program execution is difficult under the DRF0 model, because the programmer must always assume that there may have been a data race. Therefore, it may not be sufficient to reason about the execution using the intuitive sequential consistency model in order to understand and identify the error.

The recently proposed C++ memory model C++0x [7] is based on the DRF0 model and shares these shortcomings. The Java memory model [28] addresses the first problem above by providing a semantics for racy programs which is weaker than SC but still strong enough to ensure a useful form of safety. However, this weaker semantics is subtle and complex, so the debuggability problem described above is not greatly improved. Further, proving the correctness and safety of various compiler and hardware optimizations under this memory model continues to be a challenge [10, 35].

Some researchers have proposed the use of dynamic data-race detection to halt execution when it would become undefined by the memory model [2, 6]. This approach would resolve the problems with the DRF0 memory model, since a program execution would be guaranteed to be SC unless the execution is halted. However, to be useful such detection must be precise, neither allowing a program to complete its execution after a data race nor allowing a race-free execution to be erroneously rejected. Precise data-race detection in software is very expensive even with recently proposed optimizations [14], and hardware solutions [2, 30] are quite complex.

1.1 The DRFx Memory Model

In this paper we introduce the DRFx memory model, which provides a simple and strong guarantee to programmers while supporting many standard compiler and hardware optimizations. We take inspiration from the observation of Gharachorloo and Gibbons [16] that to provide a useful guarantee to programmers it suffices to detect only the data races that cause SC violations, and that such detection can be much simpler than full-fledged race detection.

The DRFx model introduces the notion of a dynamic *memory model (MM) exception* which halts a program's execution. DRFx guarantees two key 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 sequential consistency is violated in an execution of P, then the execution eventually terminates with an MM exception.

These two properties allow programmers to safely reason about *all* programs, whether race-free or not, using SC semantics: any program's execution that does not raise an MM exception is guaranteed to be SC. On the other hand, if an execution of P raises an MM exception, then the programmer knows that the program has a data race.

While our Soundness guarantee ensures that an SC violation will eventually be caught, an execution's behavior is undefined between the point at which the SC violation occurs and the exception is raised. The DRFx 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 DRFx

Gharachorloo and Gibbons describe a hardware mechanism to detect SC violations [16]. 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, this approach only handles hardware reorderings and does not consider the effect of compiler optimizations. As a result, their approach guarantees our DRF and Soundness properties with respect to the *compiled* version of a program but does not provide any guarantees with respect to the original *source* program [12, 16].

Our key contribution 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. Our 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. We also require each system call to be placed in its own region, which allows us to guarantee the DRFx model's 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 many 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 DRFx model's DRF and Soundness properties with respect to the original program, we show that 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 [19]. As in TM systems, both software and hardware conflict detection mechanisms can be considered for supporting DRFx. In this paper, we pursue a hardware detection mechanism, since the required hardware logic is fairly simple and is similar to that in existing bounded hardware transactional memory (HTM) implementations such as Sun's Rock processor [13]. In fact, our hardware design can be significantly simpler than that of a TM system. Unlike TM hardware, which needs the complex support for versioning and checkpointing to enable rollback on a conflict, a DRFx hardware only needs support for raising an exception on a conflict. Also, a DRFx 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 4 that allows us to regain most of the lost optimization potential.

1.3 Contributions

This paper makes the following contributions:

- We define the DRFx memory model for concurrent programming languages via three simple and strong guarantees for programmers (Section 2). We describe a set of conditions on a compiler and hardware design that are sufficient to enforce the DRFx memory model.

- We present a formalization of the DRFx memory model as well as of our compiler and hardware requirements (Section 3). We have proven that these requirements are sufficient to enforce DRFx.
- We describe a concrete compiler and hardware instantiation of the approach (Section 4) and have implemented a DRFx-compliant compiler by modifying LLVM [24]. We discuss an efficient solution for bounding region sizes so that a processor can detect conflicts using finite hardware resources.
- We evaluate the performance cost of our compiler and hardware instantiation in terms of lost optimization opportunity for programs in the Parsec benchmark suite (Section 5). We find that the performance overhead is on average 3.25% when compared to the baseline fully optimized implementation.

2. Motivation and Overview

This section motivates the problem addressed in the paper 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 [1, 17, 28]. Consider the example in Figure 1(a) described above. Thread `t` uses a Boolean variable `init` to communicate to thread `u` that the object `x` is initialized. Note that although 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 writes to two different memory locations. However, this reordering introduces a null dereference (and violates SC) in the interleaving shown in Figure 1(b). 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 [1]. 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 these variables as "synchronization". As such, (many) compiler and hardware reorderings are restricted across these accesses, and concurrent conflicting accesses to these variables do not constitute a data race. As a result, the revised program shown in Figure 1(c) is data-race-free and 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 proposed C++ memory model [7] 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, which we argue is an unacceptable burden. As an example, consider the program in Figure 3(a) 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.

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 [2, 6]. Note that it is not sufficient to simply detect executions that encounter a racy state as defined in Section 2.1. While the existence of such an execution implies the existence of a data race in the program, other executions can also be racy and can suffer from SC violations. Informally, an execution is considered racy if it has two conflicting accesses that are not properly synchronized, regardless of how "far away" they are from one another.

The notion of a racy program execution is made precise by Lamport's happens-before constraints [22], which define a partial order on the operations in a concurrent execution. Operations within the same thread are totally ordered by their program order. In addition, synchronization operations on the same synchronization variable induce inter-thread happens-before edges, depending on the order in which the threads perform these operations. An execution is considered to be racy if two conflicting operations are not ordered in this partial order.[1] For example, consider the interleaving in Figure 3(b). Since the two threads acquire different locks, there are no happens-before edges between operations belonging to different threads. Therefore, the conflicting accesses to `init` and `x` each constitute a race.

Precise dynamic data-race detection algorithms typically use vector-clocks [22] to reason about the happens-before ordering during executions. These algorithms are inherently costly, as they require maintaining metadata per memory location and updating this metadata at each memory access. Furthermore, an access may participate in a data race with an access that occurred arbitrarily far in the past, foiling attempts to discard metadata as the execution proceeds. Despite years of research, efficient and precise dynamic data-race detection in software has not been achieved. Even after recent optimizations, software-based data-race detection slows down the execution of the program by a factor of 8 or more [14]. Proposed hardware mechanisms can be more efficient but are complex and require significant changes to existing architectures [2, 30]. Also, hardware schemes cannot easily detect "far away" races since such schemes are limited by bounded hardware resources [30].

[1] A program can then be considered to have a data race if it has an SC execution that is racy. One can show that this definition is equivalent to our simpler definition from Section 2.1, which is more convenient to use in our formalism (Section 3).

```
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)

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

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

Figure 1. (a) Original program. (b) Transformed program. (c) Data-race-free program.

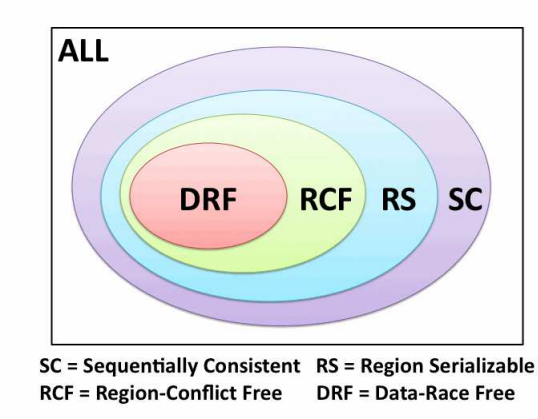

Figure 2. The relationships among various properties of a program execution.

2.5 Detecting SC Violations is Enough

The DRFx model is inspired by the observation that full happens-before data-race detection is unnecessary [16]. While such detection can be quite useful for debugging purposes, it is overly strong if our goal is to ensure that executions are SC. For example, even though the interleaving in Figure 3(b) contains a 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 3(c) do in fact violate SC and should be halted with a memory model (MM) exception.

The Venn diagram in Figure 2 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 DRF and Soundness properties described in Section 1, we must accept all executions in `DRF` and terminate all executions that are not in `SC`. However, our model allows flexibility for executions that are not `DRF` but are `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. As we describe below, this flexibility allows for a much more efficient detector than full-fledged race detection.

Our memory model only guarantees that non-SC executions *eventually* terminate with an exception. This allows us to perform SC detection *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 DRFx Model

The key idea behind enforcing the DRFx 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. We require each synchronization operation and each system call 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 DRFx 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 3(b), no regions execute concurrently and thus the DRFx runtime will not throw an exception, even though the execution contains a data race. On the other hand, in the interleaving shown in Figure 3(c), the conflicting regions {B,C} and {F,G} do execute concurrently, so an MM exception will be thrown.

2.7 From Region Conflicts to DRFx

The Venn diagram in Figure 2 illustrates the intuition for why our compiler and hardware design satisfies the DRFx properties. If a program execution is data-race-free (DRF), then concurrent regions will never conflict during that execution, i.e., the execution is *region-conflict free* (RCF). 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. 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. This establishes the DRF property of the DRFx model.

On the other hand, suppose that an execution is not SC. Then as the Venn diagram shows, that execution is also not region-conflict free, so an MM exception will be signaled. Again this property holds even in the presence of non-speculative intra-region optimizations. Therefore the Soundness property of the DRFx model is enforced.

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 our notion of region-conflict detection is *just right* to satisfy the two main DRFx 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 3(b). On the other

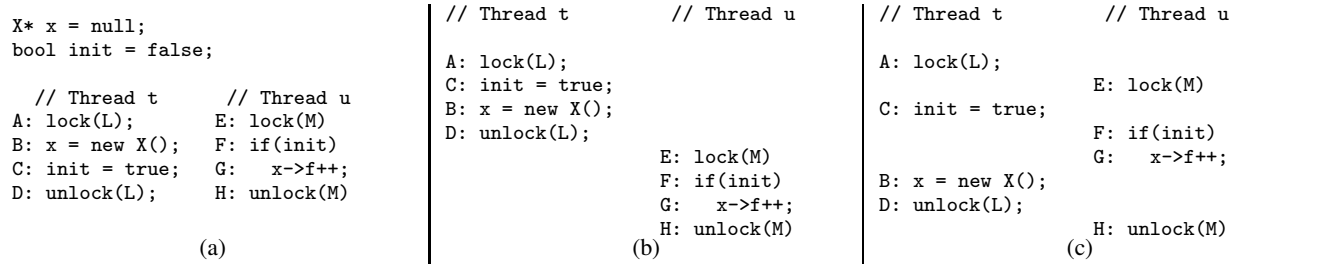

Figure 3. (a) Program with a data race. (b) Interleaving that does not expose the effect of a compiler reordering. (c) Interleaving that does.

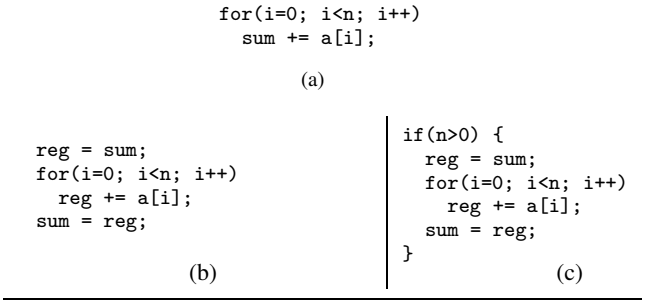

Figure 4. A transformation that introduces a read and a write.

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(a). 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 program.

Many traditional compiler optimizations (constant propagation, common subexpression elimination, dead-code elimination, etc.) satisfy the constraints above and are thus allowed by the DRFx model. Figure 4 describes an optimization that is disallowed by our model. Figure 4(a) shows a loop that accumulates the result of some computation in the `sum` variable. A transformation that allocates a register for this variable is shown in Figure 4(b). The variable `sum` is read into a register at the beginning of the loop and written back at the end of the loop. However, on code paths in which the loop is never entered, this transformation introduces a spurious read and write of `sum`. While such behavior is harmless for sequential programs, it can introduce a race with another thread modifying `sum`. One way to avoid this behavior is to explicitly check that the loop is executed at least once, as shown in Figure 4(c). The DRFx model allows the transformation with this modification, although our current compiler simply disables the transformation. In spite of this, the experimental results in Section 5 indicate that the performance reduction due to lost optimization potential is quite reasonable, on average 3.25% on our 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. Sun's TM support in the Rock processor has demonstrated that conflict detection is feasible in hardware [13]. DRFx hardware can actually be simpler than TM hardware, as we do not require speculation support. Further, unlike in a TM system, the DRFx 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.

3. Formal Description of DRFx

In this section we describe our formalization of the DRFx model. We introduce preliminary notation and definitions in Section 3.1. Section 3.2 formally presents the requirements that DRFx places on the compiler and establishes two key lemmas relating a source program to the output of a DRFx-compliant compiler. In Section 3.3 we formalize the responsibilities of the execution environment and establish two important properties of a DRFx-compliant execution. Finally, Section 3.4 uses these results to establish the properties of the DRFx model. We omit full proofs here but have made them available in a companion technical report [29].

3.1 Preliminary Definitions

A program P is a set of threads $T_1, T_2, \cdots, T_n$ 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
- a FENCE instruction 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). A FENCE instruction behaves as a no-

op: $(M, \text{FENCE}) \longrightarrow_T (M, I)$ where I is the next instruction in program order in T.

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:

$$(M, \{I_1, \cdots, I_j, \cdots, I_n\}) \longrightarrow_P (\hat{M}, \{I_1, \cdots, \hat{I_j}, \cdots, I_n\})$$

if and only 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:

$$(M, \{I_1, \cdots, I_n\}) \longrightarrow_P^* (\hat{M}, \{\hat{I_1}, \cdots, \hat{I_n}\}).$$

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 $(M_0, \{I_{10}, \cdots, I_{n0}\}) \longrightarrow_P^* (M, \{\text{END}, \cdots, \text{END}\})$ 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 $(M_0, \{I_{10}, \cdots, I_{n0}\}) \longrightarrow_P^* (M, \{I_1, \cdots, I_n\})$ 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 not an atomic operation. 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, $(M_0, \{I_{10}, \cdots, I_{n0}\}) \longrightarrow_P^* (M, \{I_1, \cdots, u, \cdots, v, \cdots, I_n\})$ where u and v are conflicting accesses. We shall say that such a partial abstract sequential execution exhibits a data race.*

3.2 DRFx-compliant Compilation

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,
- $I_k \neq I_1$ for each $2 \leq k \leq n$,
- $I_2, \cdots, I_{n-1} \in R$, 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 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 $(M_0, \{R_{10}, \cdots, R_{n0}\}) \longrightarrow_P^* (M, \{R_1, \cdots, R_n\})$ 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 our 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, $P = \{T_1, \cdots, T_n\} = \{R_{11} \cdots R_{1m_1}, \cdots, R_{n1} \cdots R_{nm_n}\}$. 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, $P' = \{R'_{11} \cdots R'_{1m_1}, \cdots, R'_{n1} \cdots R'_{nm_n}\}$.

We consider such a compilation to be DRFx*-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.

Intuitively, the above definition of a DRFx-compliant compilation requires that a DRFx-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 FENCE instructions to demarcate region boundaries. These FENCE instructions communicate the thread partitions chosen by a DRFx-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 DRFx-compliant compilations.

Lemma 1. *If $P \curvearrowright P'$ is a DRFx-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 DRFx-compliant compilation and P' has a data race, then P has a data race.*

Proof Sketch. Essentially, we take a partial abstract sequential execution of P' that exhibits the data race and reorder the trace maintaining program dependencies to achieve a trace with a region-sequential prefix and a suffix containing the race. The reordering relies on (C1). We then use (C2) and (C3) to construct an abstract sequential execution of P exhibiting a data race. □

3.3 DRFx-compliant Execution

We now formally specify the requirements that the DRFx model places on a machine executing a program. We will represent a (partial) *relaxed execution*, E, of a program as a 5-tuple $E = (M_0, \mathcal{T}, \text{EO}, \text{FO}, err)$ where M_0 is the initial machine state, $\mathcal{T}$ is the set of individual thread traces ($\mathcal{T} = \{dT_1, \cdots, dT_n\}$), EO

is a relation on operations that specifies the order in which each pair of conflicting operations occurs, FO is a global, total order on FENCE operations, and 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. We require that EO and FO are each consistent with each thread trace. In particular, if $u <_{\text{EO}} v$ for two operations u and v in the same thread trace, then u must occur before v in that thread trace, and similarly for each pair $f <_{\text{FO}} f'$ from the same thread trace.

We say that an execution $E = (M_0, \mathcal{T}, \text{EO}, \text{FO}, err)$ is *well-formed* for a program P if each operation in a thread trace dT_i is an operation from thread i in P and each thread trace dT_i satisfies intra-thread data, control, and fence dependencies. The fence dependencies ensure that all operations program-ordered before a FENCE complete before it, and all operations program-ordered after a FENCE complete after it. We model all of these dependencies as a partial order D on operations from the same thread trace. Well-formedness also requires that $\text{EO}_{|\text{wr}} \cup \text{D}$ is acyclic, where $\text{EO}_{|\text{wr}}$ is the subset of EO containing only write-to-read (i.e. read-after-write, or true) dependencies ($u <_{\text{EO}_{|\text{wr}}} v \iff u <_{\text{EO}} v \wedge u$ a write $\wedge\, v$ a read). This ensures that E has a unique, well-defined behavior $M_0 \rightsquigarrow M$.

Before defining the restrictions we place on a relaxed execution, we define an order on memory accesses that is derived from E. For a memory access $u \in dT_i \in \mathcal{T}$, define $\texttt{postFence}(u)$ to be the closest FENCE operation that executed after u in dT_i, or $\emptyset$ if no such operation exists. We use this notion to induce an order on memory accesses based on the fence order FO. Two memory accesses are weakly-fence-ordered, $u <_{\text{WFO}} v$, if $\texttt{postFence}(u) \neq \emptyset$ and either $\texttt{postFence}(u) \leq_{\text{FO}} \texttt{postFence}(v)$ or $\texttt{postFence}(v) = \emptyset$.

We also define an operator on a partial relaxed execution that truncates incomplete thread traces to their most recent FENCE operation, removes pairs from EO if at least one operation in the pair has been truncated from its thread trace, and sets err to $\emptyset$. We notate this as follows:
$\lfloor (M_0, \mathcal{T}, \text{EO}, \text{FO}, err) \rfloor = (M_0, \lfloor \mathcal{T} \rfloor, \lfloor \text{EO} \rfloor, \text{FO}, \emptyset)$

We call an execution $E = (M_0, \mathcal{T}, \text{EO}, \text{FO}, err)$ DRFx-compliant if E satisfies one of the following conditions, which formalize our notion of region-conflict detection:

(E1) $err = \emptyset$ and for all operations u and v, $u <_{\text{EO}} v \Rightarrow u <_{\text{WFO}} v$ *or*

(E2) All of the following conditions hold:

- $err = u <_{\text{EO}} v$
- u and v are from different threads
- $\texttt{postFence}(u) = \emptyset$ and $\texttt{postFence}(v) = \emptyset$
- for all operations w and z we have $w <_{\text{WFO}} z \Rightarrow z \not<_{\text{EO}} w$
- $\lfloor E \rfloor$ DRFx-compliant

We refer to a DRFx-compliant execution satisfying (E1) as *exception-free* and one satisfying (E2) as *exceptional*.

We have proven two key results for DRFx-compliant executions.

Lemma 3. *Given a well-formed exception-free* DRFx*-compliant execution $E = (M_0, \mathcal{T}, \text{EO}, \text{FO}, \emptyset)$ of a program P with valid fence-induced thread partitions, $\lfloor E \rfloor$ exhibits region-serializable behavior w.r.t. to the fence-induced partitions.*

Proof Sketch. The total order on fences FO can be viewed as a "commit" order for fence-induced regions where a region R_1 commits before R_2 if the fence following R_1 is ordered by FO before the fence following R_2. From (E1) we know that, even if portions of R_1 and R_2 executed concurrently, any conflicting accesses between them are ordered by EO in one direction, from the access in R_1 to the access in R_2. This allows us to ensure that the EO relation lifted to regions is acyclic, which implies that the execution is serializable w.r.t. to the regions. □

Lemma 4. *If there is a well-formed exceptional* DRFx*-compliant execution of a program P with valid fence-induced thread partitions, then P has a data race.*

Proof Sketch. From (E2) we have two conflicting accesses in regions that are not yet committed in the sense that no FENCE has executed after the access in either thread. Also from (E2) we can show that the execution has a region-serializable prefix. This allows us to construct an abstract sequential execution in which the two conflicting accesses are both ready to execute, thereby exhibiting a data race in P. □

3.4 DRFx Guarantees

Putting together the lemmas from Sections 3.2 and 3.3, we can prove the following theorem, which ensures that a DRFx-compliant compiler along with a DRFx-compliant execution environment enforce our DRF and Soundness properties.

Theorem 1. *If $P \curvearrowright P'$ is a* DRFx*-compliant compilation, and E is a complete* DRFx*-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.*

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 our 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.

The definition of a DRFx-compliant execution and Lemma 3 establish that all DRFx-compliant executions are region-serializable up to the latest completed region in each thread. Combining this fact with Lemma 1, we can see that, up to the completed regions, an execution is SC with respect to the original source program. Therefore, if we require that system calls are placed in their own region and that they are only passed thread-local data, we ensure that whatever behavior they exhibit would have been reachable in an SC execution of the original program. This establishes the Safety property of our DRFx model.

4. Compiler and Hardware Design

There are several possible compiler and hardware designs that meet the requirements necessary to ensure the DRFx properties as described in the previous section. In this section we describe one concrete approach, which is evaluated in the next section. Our 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 bytes it can access 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. We distinguish this from regular fences discussed in Section 3, and call the latter *hard* fences in the rest of the paper. Hard fences are necessary to respect the semantics of synchronization accesses and guarantee the properties of DRFx. 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, we observe that 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 our use of bounded-size regions.

4.1 Compiler Design

We have modified the LLVM compiler [24] to be DRFx-compliant. As specified by the requirements (C1)-(C4) in the previous section, to ensure the DRFx 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.

4.1.1 Inserting Hard Fences for DRF and Safety

A hard fence is similar to a traditional fence instruction. The hardware ensures that prior instructions have committed before allowing subsequent instructions to execute. 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 [7]), obviating the need for additional fence instructions. In our 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, we treat all `volatile` variables as atomic. All other memory operations are treated as data accesses.

To guarantee DRFx's Safety property, the compiler also inserts 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 is first 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.

To insert a hard fence, we used the `llvm.memory.barrier` intrinsic in LLVM with all of its parameters set to true. This ensures that the LLVM compiler passes do not reorder memory operations across the fence. Also, the LLVM's code generator translates it correctly to an `mfence` instruction in x86 which restricts hardware optimizations across the fence.

4.1.2 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. We employ a simple and conservative static analysis in the compiler to bound the number of memory operations in a region. While overly small regions do limit the scope of compiler optimizations, our experiments in Section 5 illustrate that the performance loss due to this limitation is about 1.7% on average. After inserting all the hard fences described earlier, the compiler performs function inlining. We then insert soft fences on the inlined code. A soft fence is conservatively inserted before each function call and return, and before each loop back-edge. Finally, we insert 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, since a soft fence is inserted at each back-edge. However, we could apply a transformation analogous to loop tiling which has 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 us to safely perform compiler optimizations across each block of R/L iterations.

4.1.3 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. As such, in our current implementation, we explicitly disable all speculative optimizations in LLVM.[2] We note, however, that there are several useful speculative optimizations that have simple variants that would be allowed by our model. For example, instead of inserting a speculative read, the compiler could insert a special prefetch instruction which the hardware would not track for purposes of conflict detection. The Itanium ISA has support for such speculation [38] in order to hide the memory latency of reads. Also, as shown earlier in Figure 4, loop-invariant code motion is allowed by our model, as long as the hoisted reads and writes are guarded to ensure that the loop body will be executed at least once.

4.2 Hardware Support

The hardware support required for DRFx is similar to conflict detection in hardware transactional memory (HTM) systems, such as Sun's Rock processor [13]. Our basic hardware design adapts the lazy conflict detection approach of Hammond et al. [18]. Our support is in some ways simpler than that of HTM, since we throw an exception upon detecting a conflict rather than performing rollback and re-execution. On the other hand, our conflict detection must be precise since a false conflict would result in a false MM exception, while a false conflict in an HTM system is acceptable as it simply causes an unnecessary re-execution.

4.2.1 Basic Hardware Design

HTM systems commonly detect conflicts by maintaining read and write access bits per cache line. However, this approach can lead to false conflicts if two different memory words accessed concurrently in different threads happen to be assigned to the same cache line. To address this problem, we track read and write accesses of a region using a circular queue buffer that we call a *region buffer* and employ lazy conflict detection.

At the beginning of a region, a region-start record is inserted into the region buffer along with the current timestamp of the processor (obtained by executing the x86 `RDTSC` instruction). On committing a read or a write operation, the processor core inserts an entry into the buffer consisting of the memory operation's address along with a bit indicating whether the operation was a read or a write, and three bits indicating the number of bytes accessed by the operation (for a total of 68 bits per buffer entry). At the end of a region, a region-end log is inserted into the region buffer along with the current timestamp.

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

We perform conflict detection *lazily*, only once when all of a region's instructions have completed. At that point, the processor core P broadcasts the region's read/write sets along with the start and end timestamps to all the other processor cores. Each remote processor core checks if there is any read-write or write-write conflict with its region buffer. On detecting a conflict, an MM exception is thrown. Otherwise, each core sends an acknowledgment to P. On receiving acknowledgments from all the processor cores, P *commits* its region by clearing its entries in the region buffer.

Because we detect conflicts lazily, it is possible that the execution is in a non-SC state for some time before the violation is detected. This could lead to two problems which we need to address. First, it is possible for a non-SC execution to cause an exception other than an MM exception (e.g., a null dereference) to be thrown, halting the program before we detect the conflict and violating the DRFx's Soundness property. To prevent this problem, the hardware performs conflict detection immediately when an instruction in a region encounters an exception. If a conflict is detected, then an MM exception is thrown instead of the original exception. Otherwise, the execution is SC and we can safely throw the original exception.

Second, a non-SC execution might enter a non-terminating loop that is not possible in an SC execution. Therefore, the hardware eagerly performs conflict detection if a region has executed for more than some constant C cycles. If a conflict is detected, an MM exception is thrown. Otherwise, execution continues as usual and the hardware waits another C cycles or until the end of the region, whichever comes first, to perform conflict detection. Note that our use of bounded regions does not solve this problem, as we only bound the number of memory operations performed rather than the number of instructions.

4.2.2 Handling Hard and Soft Fences

The current region ends whenever either a hard or soft fence instruction is encountered. If the fence terminating a region is a hard fence, the processor core that executed the region stalls until all the memory operations executed as part of the region completes, which requires it to wait for all the outstanding coherence operations to finish. Once they finish, conflict detection is initiated as described above.

The hardware cannot safely reorder memory operations across hard fences since that could violate the semantics of a synchronization access and introduce a false race. However, the hardware *can* safely reorder memory operations across regions delimited by a soft fence. Because the two adjacent regions have no synchronization accesses or system calls, it is impossible for such reordering to introduce a false race: if a reordering causes a conflict with a concurrent region on another processor, then the original program must have a data race.

Therefore, on encountering a soft fence the processor immediately continues its execution of the next region without stalling. When all the outstanding coherence operations for the first region finish, its end time is recorded in the region buffer, and then the processor core follows the same protocol as described earlier to commit that region. To commit a region, a processor core deletes all the entries corresponding to that region from its region buffer, making space for recording future memory operations.

A processor core can only run out of region buffer space due to the overlapped execution of multiple regions, since our approach to bounding regions prevents conflict detection of any single region from overflowing hardware resources. When this situation is encountered, the core simply has to stall and wait for the earliest uncommitted region to complete, which will happen once all outstanding coherence operations for that region are finished.

In this way, the hardware can overlap the execution of most regions, achieving performance close to that of DRF0 relaxed memory models while maintaining the DRFx guarantees.

4.2.3 Optimizing Conflict Detection

There are opportunities for optimizing the above hardware design in several ways. First, we can avoid inserting the same address redundantly into the region buffer. This can be achieved by keeping track of a read conflict bit and a write conflict bit for each byte in each processor core's L1 cache. Second, we can use a read and a write bloom filter [34] to accelerate conflict detections. To commit a region, only the bloom filters' signatures are broadcasted to the remote processors. On detecting a potential conflict, a precise conflict detection is initiated to avoid false conflicts. Third, if there were no cache miss, cache evictions, and coherence downgrade requests (invalidate or request for shared access) during the execution of a region, then it implies that no other processor core has a conflict. Therefore, it is safe to commit that region without checking for conflicts with other processors. We expect this to be a common case.

5. Evaluation

We implemented a DRFx-compliant compiler using LLVM [24], evaluated the compiler with programs from the Parsec [4] benchmark suite, and developed a profiling tool using Pin [27] to study the properties of regions created by this compiler. We used Parsec's `sim-large` input set for our evaluation, and each program was configured to use four threads. To perform our experiments, we used a 64-bit machine with Intel Core-2 2.40 GHZ Quad CPU, 4MB cache size, and 4GB RAM. For each experiment, we executed each program 35 times and took an average of their execution times.

Figure 5 shows the performance overhead due to several restrictions imposed on the compiler and hardware optimizations. The overheads are normalized to the performance of a binary that is compiled with the unmodified LLVM compiler at `-O3` optimization level.

The first overhead we study is due to disabling compiler optimizations and hardware reorderings when we insert hard fences for synchronization calls and system calls but without soft fences. The overhead includes that from restricting the scope of optimizations as well as from turning off all speculative optimizations done in LLVM. This experiment measures the overhead we would incur if we had "ideal" hardware with support for unbounded regions. We can see that the compiler cost due to hard fences is about 0.3% on average (maximum 2.53% for `facesim`). When we add the hardware cost for executing hard fences, the average overhead increases to about 1.55%.

Next we measured the cost of inserting soft fences to bound the region sizes. We used the static analysis described in Section 4 with a bound of 512 memory operations per region, which would require each processor core to have a region buffer of roughly 4KB or larger (512 entries * 68 bits/entry). Bounding the regions using soft fences incurs an additional 1.7% overhead which is caused by restricting compiler optimizations to smaller regions. The soft fences were translated to `nops` in the compiled binary and thus did not restrict hardware optimization.

To measure the importance of distinguishing soft fences, we also evaluated a version that executes soft fences as hard fences (using `mfence`) on our Intel Core-2 machine. This version suffered a significant performance degradation, about 92% on average (maximum is 145.5% for `facesim`). The main reason for this overhead is that the hardware is now restricted from reordering and overlapping memory operations from soft-fence delimited regions. However, as discussed in Section 4.2.2, we can indeed allow hardware to optimize across soft fences. Thus, if we have efficient hardware

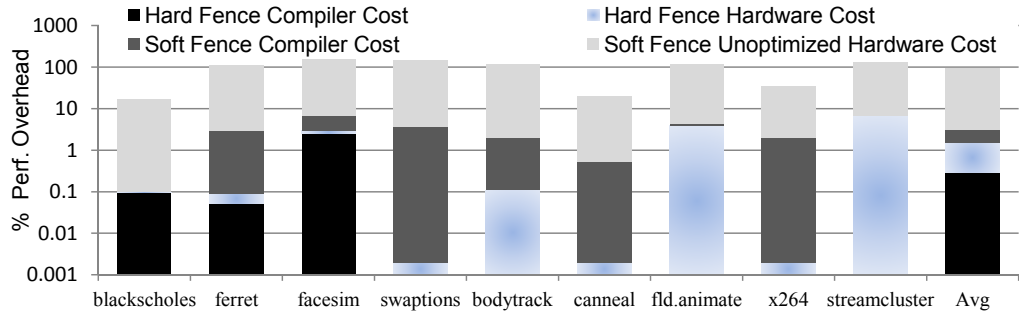

Figure 5. Performance overhead with respect to the performance of an uninstrumented binary, compiled with all the traditional compiler optimizations enabled.

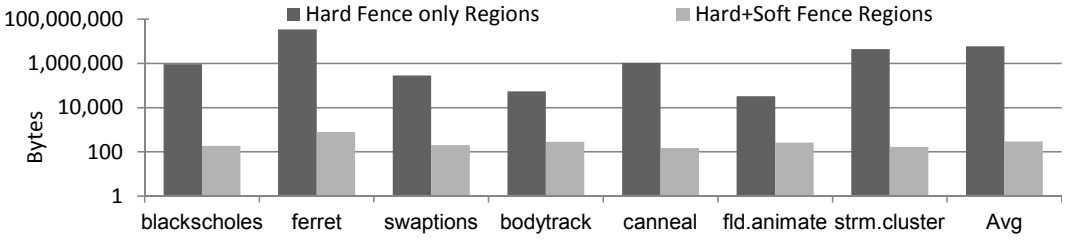

Figure 6. Maximum number of unique memory bytes accessed in any region.

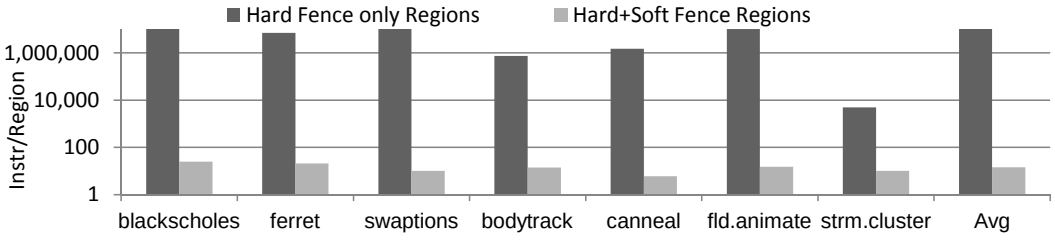

Figure 7. Average number of instructions executed in a region.

support for conflict detection, we expect that the only main sources of performance overhead in DRFx would be the compiler and hardware cost due to hard fences (1.55%), plus the compiler cost of soft fences (1.7%) which sums up to a total of about 3.25% overhead on average.

Figure 6 shows the maximum number of unique memory bytes (*footprint*) accessed in any region. The graph shows results for two configurations. One is for binaries that contains only hard fences, with no soft fences inserted, while the other is for binaries that additionally contain soft fences used to bound regions. As we can see, without region bounding there can be regions that access over 34 million unique memory bytes (`ferret`), and the average footprint of a region (not shown in the figure) is as high as about 800,000 memory bytes for `blacksholes`. With soft fences inserted, the maximum footprint is bounded within 776 memory bytes (`ferret`) (the upper limit was set to 512*8 bytes in the static analysis). The large size of unbounded regions implies that hardware detection with unbounded regions is likely to be inefficient.

Figure 7 shows the average number of instructions executed in a region. With soft fences inserted, the regions are much smaller, as expected. While soft fences allow for bounded conflict detection, the hardware may still freely reorder instructions across them. The hardware may not reorder instructions across hard fences, but as we can see the number of instructions between consecutive hard fences is sufficiently large to allow for memory-level parallelism close to that of a relaxed memory model implementation.

6. Related Work

6.1 Reducing the Cost of Sequential Consistency

Weak memory models are not necessary if both the compiler and the hardware can guarantee SC without prohibitive performance cost. Prior work has explored this possibility.

Several static analyses insert fences in a program to guarantee SC. Shasha and Snir proposed the *delay sets* algorithm for this purpose [36]. Krishnamurthy and Yelick [21] proved that computing a minimal delay set (i.e., set of fences) for a program is NP-complete. Two recent projects, Titanium [20] and Pensieve [37], extend the delay set algorithm to reduce the number of fences needed to guarantee SC. These analyses leverage a number of techniques to determine whether a memory location can potentially be involved in a race, including sharing inference [25], pointer alias analysis, and thread escape analysis. These techniques require fairly-complex whole-program analyses that are difficult to scale to large programs, especially for languages like C++. In contrast, our DRFx model allows the compiler and hardware to freely perform sequentially valid reorderings (other than speculative accesses) within a region (in addition, hardware can optimize across regions delimited by soft fences) without requiring any additional static analysis.

On the other hand, DRFx only guarantees SC for data-race free programs.

At the hardware level, various forms of speculation have been proposed to reduce the performance overhead of SC [5, 11, 33]. Of course, these techniques can only guarantee SC of the compiled program and cannot detect the non-SC behavior introduced by the compiler. Recent work on the BulkCompiler [3] addresses this problem in the context of Java programs that use locks. Even then, all these hardware proposals above require speculative execution, checkpointing, and rollback in case of conflicts, which tremendously increases the hardware complexity. Unlike ours, these proposals require possibly unbounded resources and thus have to include appropriate mechanisms to handle overflow cases.

6.2 Support for a Memory Model Exception

Adve et al. [2] proposed to detect data races at runtime using hardware support. Recently, Boehm [6] 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 [14] or incurs significant hardware complexity [30, 32] despite many proposed optimizations to the basic technique. The large overhead comes from the need to dynamically build the *happens-before* relation [22] between pairs of memory operations. Furthermore, when a memory operation occurs, it may need to be compared with other memory operations that occurred arbitrarily "far" in the past (which means that a hardware detector would have to somehow maintain information for evicted cache blocks as well). In contrast, the hardware support required for DRFx is much simpler as it requires only that we maintain a set of accesses per region and only compare two regions' sets if the regions execute concurrently. The complexity is further simplified by ensuring that the sizes of regions are bounded.

Our work builds on that of Gharachorloo and Gibbons [16], who recognized that it suffices to detect SC violations directly rather than data races. They describe a simple conflict detection algorithm that ensures our DRF and Soundness properties, but only with respect to the compiled version of a program. Their detection is not sufficient to guarantee SC in terms of the original program, because it ignores the effects of possible compiler reorderings [12, 16]. We extend their approach with a notion of regions to safely allow such compiler reorderings while still detecting all SC violations.

In concurrent work to ours, Lucia et al. [26] have also proposed a hardware exception mechanism to simplify memory consistency models for programming languages. Lucia et al. ensure a stronger property than SC, namely atomicity of *synchronization-free regions*, which are maximal regions of code delimited by synchronization operations. This property can be quite useful for understanding and debugging concurrent programs. However, it introduces additional complexity for conflict detection as they have to deal with unbounded regions. Also, conflicts must be caught as soon as they occur to prevent non-SC state being exposed to system calls. Finally, like DRFx, they too have to avoid false conflicts. Performing precise and eager conflict detection at byte granularity for unbounded-size regions is arguably more complex than our lazy conflict detection with bounded regions. We achieve efficiency in spite of smaller bounded regions by distinguishing soft fences from hard fences and allowing the hardware to optimize across soft fences.

6.3 Data-Race Freedom by Construction

Rather than signaling a run-time memory model exception, the limitations of the DRF0 model could be resolved by preventing racy programs from being written in the first place. Several static type systems have been proposed for this purpose (e.g., [8, 9, 15]). While type systems provide a useful discipline on programmers to ensure race-freedom, they typically only account for lock-based synchronization and will reject race-free programs that use other synchronization mechanisms. Further, many correct programs that employ locks will be conservatively rejected due to imprecise information about pointer aliasing. More precision in static race detection can be achieved through interprocedural analysis [31], but such whole-program analyses tend to be heavyweight.

6.4 Transactional Memory Systems

Hammond et al. [18] proposed a memory consistency model based on a transactional programming model [19]. In their approach, the programmer and compiler cooperate to ensure that each instruction is part of some transaction. The hardware then ensures that each transaction executes atomically, which in turn guarantees SC. This approach is applicable for programs written using explicit transactions, whereas DRFx is useful for programs written using locks and other traditional forms of synchronization.

Our hardware conflict detection algorithm is similar to the one proposed by Hammond et al. [18] but is simplified in a few ways. First, transactions require additional runtime support for versioning and rollback, which adds overhead and is difficult across system events such as I/O. Second, because programmers define their own transactions, the system cannot bound their size, whereas regions in DRFx are constructed by the compiler and so are easily bounded. However, transactional memory systems can incur false conflicts at the expense of extra overhead, while conflict detection in the DRFx model must be precise, which adds some extra complexity in the hardware.

7. Conclusion

We have proposed the DRFx memory model for concurrent programming languages. Like prior data-race-free memory models, DRFx guarantees that all executions of a race-free program will be sequentially consistent. However, while data-race-free models typically give no or weaker guarantees for racy programs, DRFx 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, DRFx guarantees safety, allows compiler writers to ensure correctness of their optimizations, and also enables programmers to easily reason about *all* programs using the intuitive SC semantics.

We described an approach to ensuring the DRFx properties through a novel form of cooperation between the compiler and the hardware. The compiler partitions a program into regions and can only optimize within a region. By communicating these regions to the hardware (via fences), we can ensure that both compiler optimizations and hardware reorderings preserve SC for race-free programs. We formalized a set of requirements on the compiler and the hardware and proved that they are sufficient to obey the DRFx model. We also described a concrete instantiation of the approach, whereby the compiler creates *bounded-size* regions and employs a form of *soft* fences to allow the hardware more flexibility for performing its optimizations.

Acknowledgments

We thank the anonymous reviewers for comments that improved 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. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proceedings of ISCA*, pages 2–14. ACM, 1990.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA*, pages 234–243, 1991.

[3] 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.

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

[5] C. Blundell, M. Martin, and T. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ISCA*, 2009.

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

[7] H. J. Boehm and S. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of PLDI*, pages 68–78. ACM, 2008.

[8] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of OOPSLA*, pages 56–69. ACM Press, 2001.

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

[10] P. Cenciarelli, A. Knapp, and E. Sibilio. The java memory model: Operationally, denotationally, axiomatically. In *ESOP*, pages 331–346, 2007.

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

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

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

[14] C. Flanagan and S. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of PLDI*, 2009.

[15] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of PLDI*, pages 219–232, 2000.

[16] K. Gharachorloo and P. Gibbons. Detecting violations of sequential consistency. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 316–326. ACM New York, NY, USA, 1991.

[17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of ISCA*, pages 15–26, 1990.

[18] 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.

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

[20] 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.

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

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

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

[24] C. Lattner and V. Adve. 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, 2004.

[25] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. In *Proceedings of the Tenth International Static Analysis Symposium*, 2003.

[26] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm. Conflict exceptions: Providing simple parallel language semantics with precise hardware exceptions. In *37th Annual International Symposium on Computer Architecture*, June 2010.

[27] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, Chicago, IL, June 2005.

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

[29] 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`.

[30] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. Sigrace: signature-based data race detection. In *ISCA*, 2009.

[31] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of PLDI*, pages 320–331, 2006.

[32] 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.

[33] P. Ranganathan, V. Pai, and S. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 199–210, 1997.

[34] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable hardware memory disambiguation for high ILP processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003.

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

[36] 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.

[37] 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 the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–13, 2005.

[38] W. Triebel, J. Bissell, and R. Booth. *Programming Itanium-based Systems*. Intel Press, 2001.