University of California
Los Angeles

Simplified Semantics and Debugging of Concurrent Programs via Targeted Race Detection

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Daniel Luke Marino

2011

The dissertation of Daniel Luke Marino is approved.

Madanlal Musuvathi

Sorin Lerner

Rupak Majumdar

Jens Palsberg

Todd Millstein, Committee Chair

University of California, Los Angeles

2011

This dissertation is dedicated to my family,

especially to my parents,

for their love and support.

Table of Contents

List of Figures

List of Tables

Vita

1975	Born, La Mesa, California
1999	B.A., Computer Science & Mathematics University of California, Berkeley Berkeley, California
1999–2003	Software Engineer Eagle Research, Inc. San Francisco, California
2003–2004	Director of Applications Development YMCA of San Francisco San Francisco, California
2006–2007	Graduate Student Instructor Department of Computer Science University of California, Los Angeles
2007	M.S., Computer Science University of California, Los Angeles Los Angeles, California
2008	Research Internship Microsoft Research Redmond, Washington
2009	Symantec Outstanding Graduate Student Research Award Department of Computer Science University of California, Los Angeles

2010	Research Internship IBM T.J. Watson Research Center Hawthorne, New York
2010	Research Internship Microsoft Research Redmond, Washington
2007–2011	Graduate Student Researcher Department of Computer Science University of California, Los Angeles

Publications

J. Fischer, D. Marino, R. Majumdar, and T. Millstein. "Fine-Grained Access Control with Object Sensitive Roles." In *Proceedings of the 23rd European Conference on Object-Oriented Programming, ECOOP 2009, Genoa, Italy*, pp. 173-194. Springer, 2009.

D. Marino and T. Millstein. "A Generic Type-and-Effect System." In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, Georgia*, pp. 39-50. ACM, 2009.

D. Marino, M. Musuvathi, and S. Narayanasamy. "LiteRace: Effective Sampling for Lightweight Data Race Detection." In *Proceedings of the 2009 ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, pp. 134-143. ACM, 2009.

D. Marino, A. Singh, T.Millstein, M. Musuvathi, and S. Narayanasamy. "DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages." In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada*, pp. 351-362. ACM, 2010.

D. Marino, A. Singh, T.Millstein, M. Musuvathi, and S. Narayanasamy. "A Case for an SC-Preserving Compiler." In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, California*, pp. 199-210. ACM, 2011.

S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. "JavaCOP: Declarative Pluggable Types for Java." *ACM Transactions on Programming Languages and Systems* **32**(2), pp. 1-37. ACM, 2010.

A. Singh, D. Marino, S. Narayanasamy, T.Millstein, and M. Musuvathi. "Efficient Processor Support for DRFx, a Memory Model with Exceptions." In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, California*, pp. 53-66. ACM, 2011.

Abstract of the Dissertation

Simplified Semantics and Debugging of Concurrent Programs via Targeted Race Detection

by

Daniel Luke Marino

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2011

Professor Todd Millstein, Chair

The shared memory paradigm is the de facto standard for programming parallel server and desktop applications. In this paradigm, a program is made up of a collection of threads that cooperate to perform a task and communicate by accessing a shared memory space. In order to exhibit predictable behavior, threads sharing memory must be carefully synchronized in order to avoid leaving shared memory in an inconsistent state. A data race is a common flaw in shared memory concurrent programs which occurs when multiple threads access the same memory location without synchronization.

Data races lead to insidious bugs that are difficult to find and fix. Modern optimizing compilers and hardware further complicate the situation by exposing memory models that do not guarantee sequentially consistent semantics. In such a setting, a data race can lead to behavior that is unintuitive and difficult to explain.

Precisely detecting data races at runtime could alleviate these problems. Unfortunately, precise data race detection is prohibitively expensive. This disserta-

tion presents a variety of strategies that can be used to detect only certain races at runtime with very low overhead while still providing simple, strong guarantees to programmers of shared memory, concurrent systems.

CHAPTER 1

Introduction

It has been widely noted that concurrent programming is becoming increasingly important and prevalent [Sut05]. In the "free lunch" days of the past few decades, applications have enjoyed steady performance improvements due to the frequency scaling enabled by Moore's Law. This is no longer the case, as Moore's Law is instead being realized by cramming more processing cores onto a chip. As a result, applications need to exploit opportunities for parallelism in order to improve performance and add features.

The shared memory paradigm is the de facto standard for programming parallel server and desktop applications. In this paradigm, a program is made up of a collection of threads that cooperate to perform a task and communicate by accessing a shared memory space. In order to exhibit predictable behavior, threads sharing memory must be carefully synchronized in order to avoid leaving shared memory in an inconsistent state. The bête noire of shared memory concurrent programming is the *data race* which occurs when multiple threads access the same memory location without synchronization.

Data races often lead to insidious bugs that are difficult to isolate and fix. Even well-tested, critical code can have lurking bugs caused by data races. If one doubts the potential severity of data races, consider that races in the software for a radiation therapy machine have resulted in death and injury and that the massive 2003 blackout in the Northeastern US was in part caused by a data race

[Lev93, Pou].

Bugs that result from data races are difficult to find and fix for many reasons. To begin with, reasoning about a parallel program is inherently more complicated than reasoning about a sequential program due to the need to consider multiple flows of control. Furthermore, bugs that occur as a result of improper synchronization are often intermittent, occurring only on some interleavings of the threads. Finally, the observed program failure may occur while executing code far away from the race that caused the bug. Consider the example in Figure 1.1. It shows a program that contains a data race due to unsynchronized accesses to the shared variable `nonZeroInt`, which the programmer intends never to contain the value zero. Threads 1 and 2 both check to ensure that the variable contains a value greater than one before decrementing it. However, because no synchronization mechanism is used to ensure that the decrement occurs atomically with the check, the interleaving suggested in Figure 1.1 can occur, resulting in a zero being stored into `nonZeroInt`. Thread 3, whose code is not buggy, can then cause a divide-by-zero exception due to the data race between Threads 1 and 2. This simple example demonstrates that data races can cause subtle bugs. As programs become more complicated, the bugs caused by data races can become far more difficult to understand.

Optimizing compilers and hardware further complicate the situation by transforming programs that contain data races in ways that produce unintuitive behavior. In order to give a well-defined semantics to a concurrent program, a programming language or hardware platform must specify exactly which writes to a variable or memory location may be visible to a read that executes on a different thread or processor. This specification is known as a *memory model*. In describing the example above, we implicitly assumed that a multithreaded

```
                        int nonZeroInt = 2;
```

Thread 1	Thread 2	Thread 3 can perform
`if (nonZeroInt > 1)`		`//division by zero!`
	`if (nonZeroInt > 1)`	
	`  nonZeroInt--;`	
`  nonZeroInt--;`		
		`int tmp = 10/nonZeroInt;`

Figure 1.1: A simple data race can cause a subtle bug.

program behaves as if the instructions from all threads are executed one at a time in some interleaved order, with the instructions from each individual thread executing in the order specified by the program. We further assumed that a read of a variable sees the value written by the previous write to that variable in this interleaving. This corresponds to the memory model known as sequential consistency [Lam79], or SC, which is natural for programmers to assume.

In fact, current parallel programming languages, such as Java and C++, and modern multicore architectures provide memory models that are weaker than SC. They expose these relaxed models in order to permit common performance optimizations such as common subexpression elimination in the compiler and write buffers in the hardware. While care is taken to ensure that these optimizations are not visible to programs that are free of data races, racy programs can exhibit surprising behavior under relaxed memory models. Consider the example in Figure 1.2(a) which has data races on `init` and on `x`. If these are the only two threads in the program, it seems reasonable to believe that the dereference of `x` in line D will never cause a segmentation fault, since on all interleavings of these threads, D executes only if `x` has been initialized to a non-null value. In fact, under the memory models provided by current programming languages and

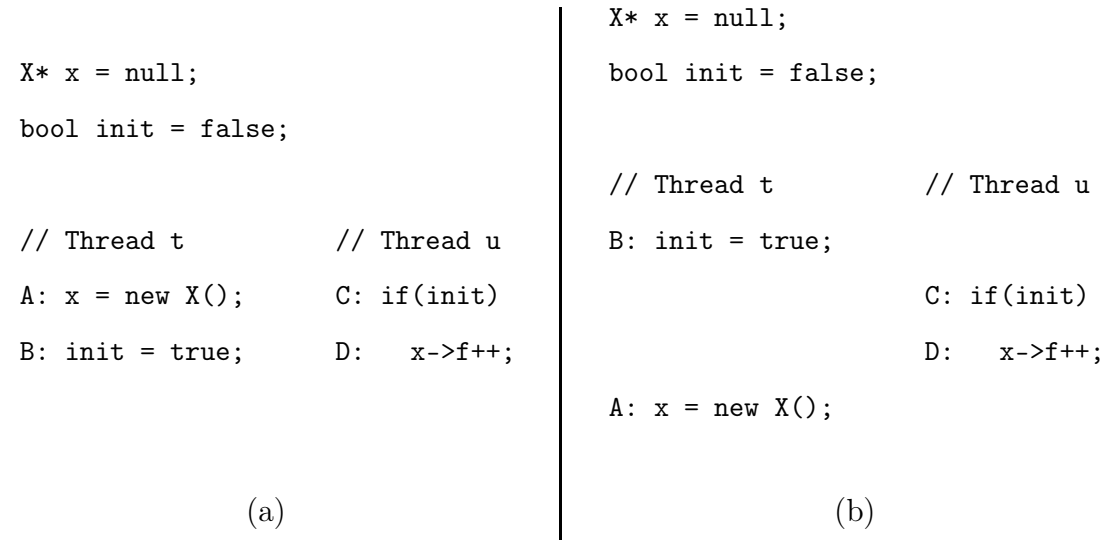

Figure 1.2: Transformation of a racy program can yield unexpected behavior. (a) Original program. (b) Transformed program.

architectures, instructions A and B may be reordered as shown in Figure 1.2(b). Thus `x` could potentially be null when it is dereferenced. Besides complicating the debugging process, the interaction between unintentional data races and optimizations can result in serious safety violations, like causing program control to jump and begin executing code at an arbitrary location in memory [BA08]. Languages generally *do* provide intuitive, SC semantics when a program is free of data races [MPA05, BA08].

Because of their serious implications, much effort has been put into research on preventing or detecting data races. While it would be desirable to prevent all data races statically, there are many practical obstacles. Most static techniques are limited only to lock-based synchronization and either greatly restrict programming style or have trouble scaling to large programs. In order to be applicable to existing programs which use a variety of synchronization primitives, the detection schemes presented in this dissertation are dynamic. Dynamic techniques are able to accurately detect races in programs that use the full range of

synchronization paradigms and can be applied equally well to large and small programs. Unfortunately, they can have a crippling effect on performance, slowing programs by 8× or more when precise detection is done in software [FF09]. Schemes for hardware-assisted dynamic data race detection lower this overhead, but require complex rollback and re-execution mechanisms in order to avoid false positives [AHM91, MSQ09].

The research in this dissertation demonstrates that dynamic data race detection can be used to improve the state of the art in shared memory concurrent programming without compromising performance and with reasonable complexity. The key is to relax the requirement that the analysis precisely identify all data races in the execution being monitored. Depending on the way in which this requirement is relaxed, we achieve race detection that solves different problems in shared memory systems without the hefty performance or complexity penalty normally associated with dynamic data race detection. This dissertation describes three instantiations of this approach. The first, LiteRace, is a testing and debugging tool. The second and third are both techniques to simplify the memory models that we expose to programmers of shared memory concurrent systems. All three support my thesis statement:

> *Although full dynamic data race detection is impractical, making carefully chosen sacrifices in detection precision enables low-overhead mechanisms that help programmers understand and debug concurrent programs.*

Chapter 2 presents some background material useful in understanding all three projects. A chapter on each project follows. Chapter 3 describes LiteRace in which an intelligent sampling technique is used to greatly reduce the overhead of traditional, precise race detection while still managing to identify 70% of the

data races exhibited during a program's execution on average. Chapter 4 describes the DRFx memory model which provides simple, strong guarantees to a programmer while still allowing most common optimizations. It relies critically on cooperation between the compiler and the hardware, on the freedom to terminate a program containing a data race with an exception, and on a form of hardware race detection which is made lightweight by ignoring data races whose accesses occur sufficiently "far apart". Despite sacrificing precision by missing certain races, the detection nevertheless provides the basis for a strong guarantee to the programmer: normally terminating programs exhibit SC behavior, while exceptional programs have a data race. Chapter 5 presents research suggesting that a compiler that preserves sequential consistency, even in the face of data races, can still output code that performs well. It achieves this using lightweight, conservative, static and dynamic techniques to establish data race freedom in areas where optimizations are applied. As in DRFx, the dynamic component is implemented in hardware but is aided by the compiler. Recovery code inserted by the compiler maintains SC in the event that a race is dynamically detected, avoiding the need to throw an exception. Finally, Chapter 6 will describe some related work and Chapter 7 concludes.

CHAPTER 2

Background

This chapter provides details on data races, synchronization, data race detection, and relaxed memory models. It provides useful context for understanding the material in the following chapters.

2.1 Data Races

When multiple threads share memory state, they must take care to ensure that invariants assumed about that state are not violated due to concurrent access to memory by multiple threads. This is done by using special synchronization operations to coordinate activities between threads. Common programming language synchronization operations include mutual exclusion locks, semaphores, and condition variables. These high-level synchronization operations are generally implemented using lower-level machine synchronization operations such as compare-and-swap. We refer to all memory accesses used to implement synchronization operations as *atomic* reads and writes.

We now present some informal definitions related to memory accesses, synchronization, and data races. The definitions assume a program made up of multiple threads that can only communicate through shared memory. They access this memory using ordinary reads and writes and atomic reads and writes which are used for the purposes of synchronization.

Definition 2.1 (Conflicting memory accesses)**.** Two memory accesses *conflict* if they access the same memory location, at least one writes to memory, and at least one is not an atomic access (i.e., part of a synchronization operation).

Definition 2.2 (Racy execution (simultaneous))**.** A program execution exhibits a data race if two threads execute conflicting memory accesses simultaneously. We call such an execution racy.

Definition 2.3 (Racy program)**.** We say that a program contains a data race if there is some execution of the program that exhibits a data race. We call such a program racy.

Definition 2.4 (Data-race-free program)**.** A program that does not contain a data race is data-race-free.

A programmer can ensure that a program is data-race-free by using synchronization operations to ensure that no two threads can access the same memory at the same time. For instance, the code in Figure 2.1 fixes the buggy code from Figure 1.1 by using a mutex lock to ensure that the threads never access `nonZeroInt` simultaneously. [1]

Definition 2.2 is quite a strict definition for a racy execution in the sense that only executions that actually perform accesses simultaneously are considered to exhibit a data race. In fact, the most commonly used definition for a racy execution is based on Lamport's happened-before relation [Lam79] and takes advantage of the fact that if synchronization operations are not used to enforce ordering between conflicting accesses on different threads, then the potential for

[1] Note that inserting sufficient synchronization to ensure data race freedom does not guarantee correct maintenance of program invariants. We could make the program from Figure 1.1 data-race-free by surrounding each individual access to `nonZeroInt` with lock and unlock operations. But, this synchronization would still lead to buggy behavior.

```
int nonZeroInt = 2;
mutex m;
```

```
//Thread 1                  //Thread 2                  //Thread 3
m.lock();                   m.lock();                   m.lock();
if (nonZeroInt > 1)         if (nonZeroInt > 1)         int tmp = 10/nonZeroInt;
  nonZeroInt--;               nonZeroInt--;             m.unlock();
m.unlock();                 m.unlock();
```

Figure 2.1: Locking can be used to ensure data race freedom, enforce atomicity, and maintain invariants.

simultaneous execution of these accesses exists. We will define the happened-before relation as follows.

Definition 2.5 (Happened-before)**.** The happened-before ($<_{\text{HB}}$) relation is a strict (irreflexive) partial order on the operations in a multi-threaded program defined by the following inductive rules.

(HB1) $a <_{\text{HB}} b$ if a and b are operations from the same sequential thread of execution and a is executed before b.

(HB2) $a <_{\text{HB}} b$ if a and b are synchronization operations on the same variable from different threads such that the semantics of the synchronization dictates that a preceded b in the execution.

(HB3) The relation is transitive, so if $a <_{\text{HB}} b$ and $b <_{\text{HB}} c$, then $a <_{\text{HB}} c$.

The informal statement of rule HB2 allows us to capture the wide range of synchronization operations that are used, both high and low level. For instance, in the case of mutual exclusion locks, we know that during a particular execution, a particular release of a mutex happened-before the following acquire of the same

mutex. As another example, we know that an atomic write happened-before an atomic read that sees the value written by that write.

Using this definition of happened-before, we formulate an alternative definition of a racy execution.

Definition 2.6 (Racy execution (happened-before))**.** An execution exhibits a data race if there are conflicting memory accesses a and b such that $a \not<_{\text{HB}} b$ and $b \not<_{\text{HB}} a$. We call such an execution racy.

Notice that Definition 2.3 for a racy program relies on the definition of a racy execution, of which we now have two. In most settings, defining a racy program using either Definition 2.2 or Definition 2.6 for a racy execution is equivalent (e.g., [BA08]). But from a dynamic detection standpoint, it is most sensible and effective to use the happened-before-based definition for two reasons:

1. More executions exhibit a race under Definition 2.6 than under Definition 2.2. Thus we are more likely to find an execution that reveals a data race in a program. This improves the effectiveness of a dynamic detection scheme.

2. Determining actual simultaneity in a complex system, such as a multicore machine, is generally not feasible.

2.2 Happened-before Data Race Detection

In order to perform dynamic, happened-before-based data race detection, a tool must keep track of the memory accesses and synchronization operations performed by each thread during an execution. It must then construct the happened-before relation for the execution, and for each pair of conflicting accesses, deter-

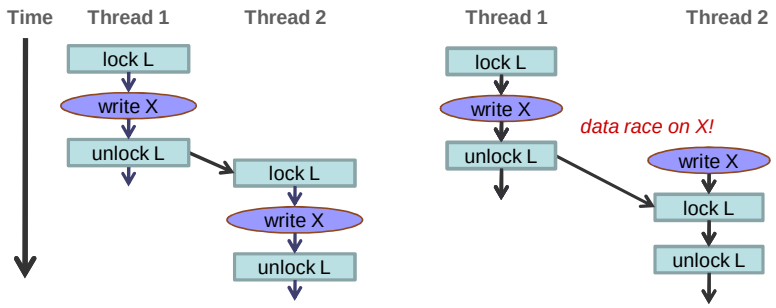

Figure 2.2: Examples of properly and *im*properly synchronized accesses to a memory location X. Edges between nodes represent a happened-before relationship. There is no data race for the example on the left, because there is a happened-before relationship (due to unlock and lock operations) between the two writes to the location X. However, for the example on the right, there is no happened-before relationship between the two writes. Thus, it has a data race.

mine whether or not one happened-before the other. A data race is found if there are two conflicting accesses neither of which happened-before the other.

Figure 2.2 shows how the happened-before relation is used to find data races. The edges between instructions indicate a happened-before relationship derived using rule HB1 or HB2. Transitively, by HB3, if there is a path between any two nodes, then there is a happened-before relationship between the two nodes. The example on the left in Figure 2.2 shows two properly synchronized accesses to a shared memory location. Since the two writes have a path between them, they do not race with each other. In the example shown on the right in Figure 2.2, thread 2 accesses a shared memory location without proper synchronization. Because there is no path between the two writes, the two writes are involved in a data race.

There are two primary sources of overhead for a happened-before-based dynamic data race detector implemented in software. One, it needs to instrument all the memory operations and all the synchronizations operations executed by the application. This results in a high performance cost due to the increase in the number of additional instructions executed at runtime. Two, it needs to main-

tain metadata for each memory location accessed by the application. Most of the happened-before-based algorithms [Lam78, Net93, AHM91, CMN91, CB01, DS90, Cru91, Sch89, PK96, RB00, MC91] use *vector clocks* to keep track of logical timestamps for all memory operations along with the addresses of the locations they accessed. Maintaining such metadata further slows down the program execution due to increased memory cost. Even using optimizations suggested in recent research, happened-before race detection performed in software slows down the execution of a program by 8× or more on average [FF09]. Meanwhile, proposals for hardware assisted happened-before-based data race detection [AHM91, MSQ09] suffer from both incompleteness and complexity. The tracking of metadata in fixed size hardware structures, such as caches, limits the window in which races are detected. Furthermore, the proposed mechanisms either detect races at the cache line granularity or by using signature based summaries, both of which lead to false positives. Thus, in order to make the detection precise, they rely on complex checkpointing schemes in order to roll back and re-execute when a potential race is encountered.

The detection schemes used in the systems presented in this dissertation avoid the cost and complexity inherent to precise happened-before-based data race detection by targeting the detection at particular goals which can be achieved while sacrificing some precision. In the case of LiteRace, described in the next chapter, the goal is finding bugs in mature applications. Chapters 4 and 5 detail approaches where the goal is a simplified memory model.

2.3 Relaxed Memory Models

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 become visible to other threads, and thereby the possible values a read can return. It provides a contract that programmers can assume and that compilers and hardware must obey.[2] 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 common optimizations, potentially harming program performance.

A case in point is *sequential consistency* (SC) [Lam79], 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 where each read from a location sees the value from the previous write to that location in the interleaving. 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 (Figure 1.2 demonstrates such a transformation). 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 order to allow these optimizations, hardware architectures and programming languages provide a variety of relaxed

[2]The term memory model is used to describe both the contract between the programmer and the programming language and the contract between the compiler (or programming language implementation) and the hardware. The compiler, then, provides certain guarantees to the programmer that it implements using the guarantees provided to it by the underlying hardware architecture.

memory models which are weaker (i.e., provide weaker guarantees) than SC.

2.4 Interaction of Memory Models and Data Races

Weak memory models allow relaxations of memory access ordering, but they must also provide mechanisms to enforce ordering when program behavior requires it. At the programming language level, this mechanism is usually provided in the form of synchronization operations (including high-level mechanisms like locks as well as individual memory accesses used for synchronization which are identified using qualifiers such as `volatile` in Java [MPA05] and `atomic` in C++ [BA08]). Hardware provides mechanisms such as memory fences and atomic operations that compilers use to implement synchronization operations.

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 [AH90], which attempts to strike a middle ground between simplicity for programmers and flexibility for compilers and hardware. In the DRF0 model, compilers and hardware are restricted from performing certain optimizations and reordering across synchronization operations while programmers are guaranteed SC behavior for all properly synchronized programs (i.e., data-race-free programs).[3] Unlike the full SC model, compilers and hardware are still able to perform common, sequentially valid optimizations in areas of code that do not contain synchronization operations.

The DRF0 model provides a simple and strong guarantee for data-race-free

[3]In DRF0 models, the definition of a data-race-free program requires the absence of a data race in all *SC executions* of the program. In this dissertation, it is also assumed that Definitions 2.2 through 2.6 need only consider sequentially consistent executions.

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 show how a sequentially valid compiler optimization can cause a program to jump to arbitrary code in the presence of a data race [BA08].

- 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 [BA08] is based on the DRF0 model and shares these shortcomings. The Java memory model [MPA05] 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.

Researchers have previously proposed the use of dynamic data race detection to halt execution when it would become undefined by the memory model [AHM91, Boe09, EQT07]. But due to the cost and complexity discussed in Section 2.2, the use of precise data race detection for the purpose of preventing memory model effects from confounding programmers is impractical, especially considering that the reason for providing a relaxed memory model is to enable perfor-

mance optimization. Chapters 4 and 5 detail novel schemes for using imprecise, hardware-assisted race detection which is targeted and efficient to achieve simplified memory models with good performance.

CHAPTER 3

LiteRace

Like most prior work in dynamic data race detection, LiteRace aims to find bugs. In particular, its goal is to find data races with very low overhead so that it can be run on a large number of executions. Ideally, the instrumentation would be so lightweight that it could be used during beta testing of a product. The prohibitive slowdown of existing detectors limits the amount of testing that can be done for a given amount of time and resources. Also, users shy away from intrusive tools that do not allow them to test realistic program executions. A second goal of LiteRace is to provide this lightweight detection while never reporting a *false* data race. Data races are very difficult to debug and triage. So false positives severely limit the usability of a tool from a developer's perspective.

As discussed earlier, precise data race detectors have an extremely high runtime overhead, slowing down applications by 8× or more on average [FF09].[1] (Intel's Thread Checker [SBM06], incurs a performance overhead on the order of 200×.[2]) Such a slowdown is unacceptable for LiteRace, given its goal of being

[1]Data race detectors based on the lockset algorithm [SBN97] or a hybrid of lockset and happened-before detection achieve better performance at the cost of precision. More details are discussed in Chapter 6.

[2]FastTrack manages to get an 8× overhead both by using novel improvements in the happened-before detection algorithm and also by targeting Java programs which can be instrumented using a specialized virtual machine. Intel's Thread Checker, on the other hand, is capable of instrumenting and finding races in arbitrary x86 binaries written in any language and using all types of synchronization.

usable during beta testing of applications. In order to achieve acceptable speed, some sacrifice will have to be made in detection precision.

Rather than analyzing every memory access in a program, LiteRace uses *sampling* to significantly reduce the cost of dynamic data race detection. While a sampling approach may seem unlikely to find many data races (after all, most memory accesses do not participate in a race and *both* racing accesses need to be analyzed), experimental results show that a carefully chosen sampling algorithm can be effective. The algorithm is based on the *cold-region hypothesis* that data races are likely to occur when a thread is executing a "cold" code region (code that it has not executed frequently). Data races that occur in hot regions of well-tested programs either have already been found and fixed, or are likely to be benign. The adaptive sampler starts off by sampling all the code regions at 100% sampling rate. But every time a code region is sampled, its sampling rate is progressively reduced until it reaches a lower bound. Thus, cold regions are sampled at a very high rate, while the sampling rate for hot regions is adaptively reduced to a very small value. In this way, the adaptive sampler avoids slowing down the performance-critical hot regions of a program.

The research presented in this chapter includes the following important contributions:

- LiteRace demonstrates that the technique of sampling can be used to significantly reduce the runtime overhead of a data race detector without introducing any additional false positives. It is the first data race detection tool that uses sampling to reduce the runtime performance cost. By permitting users to adjust the sampling rate to provide a bound on the performance overhead, LiteRace makes it feasible to enable data race detection even during beta testing of industrial applications.

- Several sampling strategies are explored. The results show that a naïve random sampler is inadequate for maintaining a high detection rate while using a low sampling rate. A more effective adaptive sampler that heavily samples the first few executions of a function *in each thread* is proposed.

- An implementation of LiteRace using the Phoenix analysis framework [Mica] is discussed. The tool was used to analyze Microsoft programs such as ConcRT and Dryad, open-source applications such as Apache and Firefox, and two synchronization-heavy micro-benchmarks. The results show that, on average, by logging less than 2% of memory operations, LiteRace can detect nearly 70% of data races in a particular execution.

The rest of this chapter is organized as follows. §3.1 presents an overview of a sampling based approach to reduce the runtime cost of a data race detector. §3.2 details the implementation of the race detector. Experimental results are presented in §3.3 and §3.4 concludes.

3.1 LiteRace Overview

This section presents a high-level overview of LiteRace. The implementation details together with various design trade-offs are discussed in §3.2.

3.1.1 Case for Sampling

The key premise behind LiteRace is that sampling techniques can be effective for data race detection. While a sampling approach has the advantage of reducing the runtime overhead, the main trade-off is that it can miss data races. This trade-off is acceptable for the following reasons. First, dynamic techniques cannot find

all data races in the program anyway. They can only find data races on thread interleavings and paths explored at runtime. Furthermore, a sampling-based detector, with its low overhead, would encourage users to widely deploy it on many more executions of the program, possibly achieving better coverage. Thus as long as the sampling technique doesn't miss too many races, it could prove useful.

Another key advantage is that sampling techniques provide a useful *knob* that allow users to trade runtime overhead for coverage. For instance, users can increase the sampling rate for interactive applications that spend most of their time waiting for user inputs. In such cases, the overhead of data race detection is likely to be masked by the I/O latency of the application.

3.1.2 Events to Sample

Data race detection requires logging the following events at runtime.

- Synchronization operations are logged along with a logical timestamp that reflects the happened-before relation between these operations.
- Reads and writes to memory are logged in the program order, logically happening at the timestamp of the preceding synchronization operation of the same thread.

These logs can then be analyzed offline or during program execution (§3.2.4). The above information allows a data race detector to construct the happened-before ordering between synchronization operations and the memory operations executed in different threads. A data race is detected if there is no synchronization ordering between two accesses to the same memory location, and at least one of them is a write.

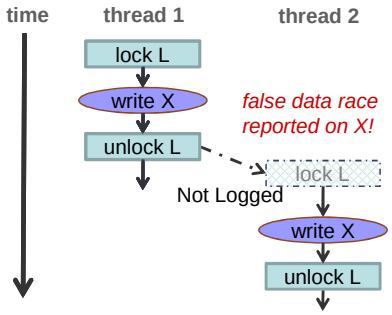

Figure 3.1: Failing to log a synchronization operation results in loss of happened-before edges. As a result, a false data race on X would be reported.

Clearly instrumenting code to log every memory access would impose a significant overhead. By sampling only a fraction of these events, overhead is reduced in two ways. First, the execution of the program is much faster because of the reduced instrumentation. Second, the data race detection algorithm needs to process fewer events making it faster as well.

While sampling can reduce runtime overhead, choosing which events to log and which events not to log must be done carefully. In particular, LiteRace must log *all* the synchronization events in order to avoid reporting false data races. Figure 3.1 shows why this is the case. Synchronization operations induce happened-before orderings between program events. Any missed synchronization operation can result in missing edges in the happened-before graph. The data race detection algorithm will therefore incorrectly report false races on accesses that are otherwise ordered by the unlogged synchronization operations. To avoid such false positives, it is necessary to log all synchronization operations. However, for most applications, the number of synchronization operations is small compared to the number of instructions executed in a program. Thus, logging all synchronization operations does not cause significant performance overhead.

Memory accesses can, however, be selectively sampled. If LiteRace chooses not to log a particular memory access, it may miss a data race involving that access (a false negative). As discussed in §3.1.1, this is an acceptable trade-off. But, a good strategy for selecting which memory accesses to log is essential in order not to miss too many races. A data race involves two accesses and a sampler needs to successfully log both of them to detect the race. A sampler that accomplishes this is described below.

3.1.3 Sampler Granularity

In LiteRace, every function body is a unit of sampling. A static instrumentation tool creates two copies for each function as shown in Figure 3.2. The instrumented function logs all the memory operations (their addresses and program counter values) and synchronization operations (memory addresses of the synchronization variables along with their timestamps) executed in the function. The un-instrumented copy of the function logs only the synchronization operations. Before entering a function, the sampler (represented as dispatch check in Figure 3.2) is executed. Based on the decision of the sampler, either the instrumented copy or the un-instrumented copy of the function is executed. As the dispatch check happens once per function call, it is essential that the dispatch code is as efficient as possible.

3.1.4 Thread Local Adaptive Bursty Sampler

There are two requirements for a sampling strategy. Ideally, a sampling strategy should maintain a high data race detection rate even with a low sampling rate. Also, it should enable an efficient implementation of the dispatch check that determines if a function should be sampled or not. A naïve random sampler does

not meet these requirements as shown in §3.3.

The LiteRace sampler is an extension of the adaptive bursty sampler [HC04], previously shown to be successful for detecting memory leaks. An adaptive bursty sampler starts off by analyzing a code region at a 100% sampling rate, which means that the sampler always runs the instrumented copy of a code region the first time it is executed. Since the sampler is bursty, when it chooses to run the instrumented copy of a region, it does so for several consecutive executions. The sampler is adaptive in that after each bursty sample, a code region's sampling rate is decreased until it reaches a lower bound.

To make the adaptive bursty sampler effective for data race detection, the above algorithm is modified to make it "thread local". The rationale is that, at least in reasonably well-tested programs, data races occur when a thread executes a cold region. Data races between two hot paths are unlikely – either such a data race is already found during testing and fixed, or it is likely to be a *benign* or intentional data race. In a "global" adaptive bursty sampler [HC04], a particular code region can be considered "hot" even when a thread executes it for the first time. This happens when other threads have already executed the region many times. LiteRace avoids this by maintaining separate sampling information for each thread, effectively creating a "thread local" adaptive bursty sampler. The experiments in §3.3 show that this extension significantly improves the detection rate.

Note that a thread-local adaptive sampler can also find some data races that occur between two hot regions or between a hot and a cold region. The reason is that LiteRace's adaptive sampler initially assumes that all the regions are cold, and the initial sampling rate for every region is set to 100%. Also, the sampling rate for a region is never reduced below a lower bound. As a result, the sampler,

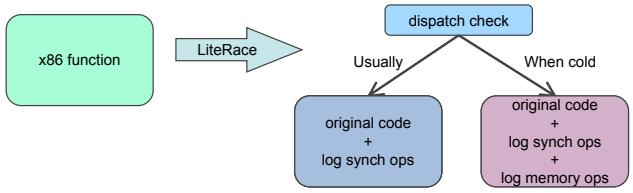

Figure 3.2: LiteRace Instrumentation.

even while operating at a lower sampling rate, might still be able to gather enough samples for a frequently executed hot region. Thus LiteRace's adaptive sampler is still able to find some data races between hot-hot regions and hot-cold regions in a program.

3.2 LiteRace Implementation

This section describes the implementation details of LiteRace.

3.2.1 Instrumenting the Code

LiteRace is based on static instrumentation of x86 binaries and does not require the source code of the program. It was built by using the Phoenix [Mica] compiler and analysis framework to parse the x86 executables and perform the transformation depicted in Figure 3.2. LiteRace creates two versions for each function: an *instrumented* version that logs all the memory operations and an *uninstrumented* version that does not log any memory operation. As explained in §3.1, avoiding false positives requires instrumenting *both* the instrumented and the uninstrumented versions to log synchronization operations. Then, LiteRace inserts a dispatch check at every function entry. This check decides which of the two versions to invoke for a particular call of the function at runtime.

In contrast to prior adaptive sampling techniques [HC04], LiteRace maintains

profiling information *per thread*. For each thread, LiteRace maintains a buffer in the thread-local storage that is allocated when the thread is created. This buffer contains two counters for each instrumented function: the frequency counter and the sampling counter. The frequency counter keeps track of the number of times the thread has executed a function and determines the sampling rate to be used for the function (a frequently executed function will be sampled at a lower sampling rate). The sampling counter is used to determine when to sample the function next. On function entry, the dispatch check decrements the sampling counter corresponding to that function. If the sampling counter's value is non-zero, which is the common case, the dispatch check invokes the uninstrumented version of the function. Once the sampling counter reaches zero, the dispatch check invokes the instrumented version of the function for the next several invocations, depending on the configured burst length. After the last invocation in the sampled burst, the dispatch check code sets the sampling counter to a new value based on the current sampling rate for the function as determined by the frequency counter.

As the dispatch check is executed on every function entry, it is important to keep the overhead of this check low. To avoid the overhead of calling standard APIs for accessing thread-local storage, LiteRace implements an inlined version using the *Thread Execution Block* [Micb] structure maintained by the Windows OS for each thread. Also, the dispatch check uses a single register `edx` for its computation. The instrumentation tool analyzes the original binary for the function to check if this register and the `eflags` register are live at function entry, and injects code to save and restore these registers only when necessary. In the common case, LiteRace's dispatch check involves 8 instructions with 3 memory references and 1 branch (that is mostly not taken). The runtime overhead of the dispatch check is measured in the experiments described in §3.3.

Table 3.1: How LiteRace logs synchronization operations.

Synchronization Op	SyncVar	Add'l Sync?
Lock / Unlock	Lock Object Address	No
Wait / Notify	Event Handle	No
Fork / Join	Child Thread Id	No
Atomic Machine Ops	Target Memory Addr.	Yes

3.2.2 Tracking Happened-before

As mentioned earlier, avoiding false positives requires accurate happened-before data. It is trivial to ensure that the happened-before relation for events of the same thread is correctly recorded since the logging code executes on the same thread as the events being logged. Correctly capturing the happened-before data induced by the synchronization operations between threads in a particular program execution requires more work.

For each synchronization operation, LiteRace logs a *SyncVar* that uniquely identifies the synchronization object and a logical timestamp that identifies the order in which threads perform operations on that object. Table 3.1 shows how LiteRace determines the SyncVar for various synchronization operations. For instance, LiteRace uses the address of the lock object as a SyncVar for `lock` and `unlock` operations. The logical timestamp in the log should ensure that if `a` and `b` are two operations on the same SyncVar and `a` $<_{\text{HB}}$ `b`, then `a` has a smaller timestamp than `b`. The simplest way to implement the timestamp is to maintain a global counter that is atomically incremented at every synchronization operation. However, the contention introduced by this global counter can dramatically slowdown the performance of LiteRace-instrumented programs on multi-processors. To alleviate this problem, LiteRace uses one of 128 counters uniquely determined by a hash of the SyncVar for the logical timestamp.

To ensure the accuracy of the happened-before relation, it is important that

LiteRace computes and logs the logical timestamp *atomically* with the synchronization operation performed. For some kinds of synchronization, the semantics of the operation can be leveraged to guarantee this. For instance, by logging and incrementing the timestamp *after* a `lock` instruction and *before* an `unlock` instruction, it is guaranteed that an `unlock` operation on a particular mutex will have a smaller timestamp than a subsequent `lock` operation on that same mutex in another thread. For wait/notify operations, LiteRace increments and logs the timestamp before the notify operation and after the wait operation to guarantee consistent ordering. A similar technique is used for fork/join operations.

For some synchronization operations, however, LiteRace is forced to add additional synchronization to guarantee atomic timestamping. For example, consider a target program that uses atomic compare-and-exchange instructions to implement its own locking. Since LiteRace doesn't know if a particular compare-and-exchange is acting as a "lock" or as an "unlock", it introduces a critical section to ensure that the compare-and-exchange and the logging and incrementing of the timestamp are all executed atomically. Without this, LiteRace could generate timestamps for these operations that are inconsistent with the actual order. This additional effort is absolutely essential in practice, and running LiteRace without this additional synchronization results in hundreds of false data races for some benchmarks.

3.2.3 Handling Dynamic Allocation

Another subtle issue is that a dynamic data race detector should account for the reallocation of the same memory to a different thread. A naïve detector might report a data race between accesses to the reallocated memory with accesses performed during a prior allocation. To avoid such false positives, LiteRace

additionally monitors all memory allocation routines and treats them as additional synchronization performed on the memory page containing the allocated or deleted memory.

3.2.4 Analyzing the Logs

The LiteRace profiler generates a stream of logged events during program execution. The current implementation writes these events to the disk and processes them offline to find data races. The main motivation for this design decision was to minimize perturbation of the runtime execution of the program. It would also be possible to use an online detector, possibly avoiding a runtime slowdown by using an idle core in a many-core processor. The logged events are processed using a standard implementation [SBM06] of the happened-before based data race detector described in §2.2.

3.3 Results

This section presents experimental results from running LiteRace. §3.3.1 describes the benchmarks and §3.3.2 explains the samplers that are evaluated. In §3.3.3, the effectiveness of the various samplers in detecting data races is explored. The results show that LiteRace's thread-local adaptive sampler achieves a high data race detection rate, while maintaining a low sampling rate. §3.3.4 discusses the performance and log size overhead of the thread-local adaptive sampler implemented in LiteRace, and compares it to an implementation that logs all the memory operations. All experiments were run on a Windows Server 2003 system with two dual-core AMD Opteron processors and 4 GB of RAM.

3.3.1 Benchmarks

The four industrial-scale concurrent programs listed in Table 3.2 were used as benchmarks. Dryad is a distributed execution engine, which allows programmers to use a computing cluster or a data center for running coarse-grained data-parallel applications [IBY07]. The test harness used for Dryad was provided by its lead developer. The test exercises the shared-memory channel library used for communication between the computing nodes in Dryad. Experiments were run with two versions of Dryad, one with the standard C library statically linked in (referred to as Dryad-stdlib), and the other without. For the former, LiteRace instruments all the standard library functions called by Dryad. The second benchmark, ConcRT, is a concurrent run-time library that provides lightweight tasks and synchronization primitives for developing data-parallel applications. It is part of the parallel extensions to the .NET framework [Duf07]. Two different test inputs for ConcRT were used: Messaging, and Explicit Scheduling. These are part of the ConcRT concurrency test suite. Apache, an open-source HTTP web server, is the third benchmark. Overhead and effectiveness of LiteRace are evaluated for two different Apache workloads (referred to as Apache-1 and Apache-2). The first consists of a mixed workload of 3000 requests for a small static web page, 3000 requests for a larger web page, and 1000 CGI requests. The second consists solely of 10,000 requests for a small static web page. For both workloads, up to 30 concurrent client connections are generated by Apache's benchmarking tool. The final benchmark is Firefox, the popular open-source web browser. The overhead and sampler effectiveness for the initial browser start-up (Firefox-Start) and for rendering an html page consisting of 2500 positioned DIVs (Firefox-Render) are measured.

Table 3.2: Benchmarks used to evaluate LiteRace. The number of functions and the binary size includes executable and any instrumented library files.

Benchmarks	**Description**	**# Fns**	**Bin. Size**
Dryad	Library for distributed data-parallel apps	4788	2.7 MB
ConcRT	.NET Concurrency runtime framework	1889	0.5 MB
Apache 2.2.11	Web server	2178	0.6 MB
Firefox 3.6a1pre	Web browser	8192	1.3 MB

Table 3.3: Samplers evaluated along with their short names used in figures, short descriptions, and effective sampling rates averaged over the benchmarks studied. The weighted average uses the number of memory accesses in each benchmark application as a weight.

Sampling Strategy	**Short Name**	**Description**	**Weighted Avg ESR**	**Average ESR**
Thread-local Adaptive	TL-Ad	Adaptive back-off per function / per thread (100%,10%,1%,0.1%); bursty	1.8%	8.2%
Thread-local Fixed 5%	TL-Fx	Fixed 5% per function / per thread; bursty	5.2%	11.5%
Global Adaptive	G-Ad	Adaptive back-off per function globally (100%, 50%, 25%, ... , 0.1%); bursty	1.3%	2.9%
Global Fixed	G-Fx	Fixed 10% per function globally; bursty	10.0%	10.3%
Random 10%	Rnd10	Random 10% of dynamic calls chosen for sampling	9.9%	9.6%
Random 25%	Rnd25	Random 25% of dynamic calls chosen for sampling	24.8%	24.0%
Un-Cold Region	UCP	First 10 calls per function / per thread are NOT sampled, all remaining calls are sampled	98.9%	92.3%

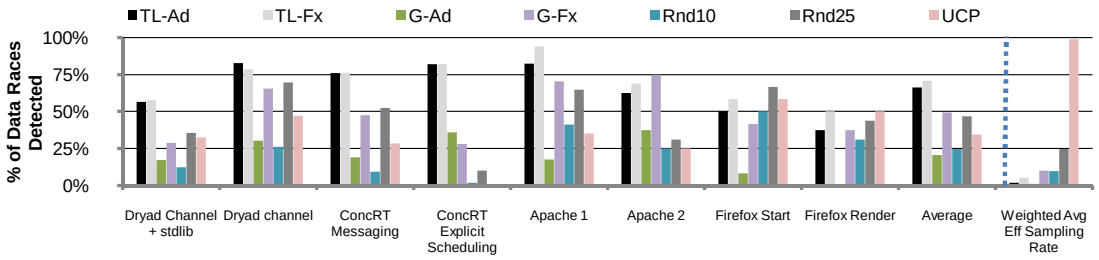

Figure 3.3: Proportion of static data races found by various samplers. The figure also shows the weighted average effective sampling rate for each sampler, which is the percentage of memory operations logged (averaged over all the benchmarks).

3.3.2 Evaluated Samplers

The samplers that are evaluated are listed in Table 3.3. The "Short Name" column shows the abbreviation that is used for the samplers in the figures throughout the rest of this section. The table also shows the effective sampling rate (ESR) for each sampler. The effective sampling rate is the percentage of memory operations that are logged by a sampler. Two averages for effective sampling rate are shown. One is just the average of the effective sampling rates over the nine benchmark-input pairs described in §3.3.1. The other is the weighted average, where the weight for a benchmark-input pair is based on the number of memory operations executed at runtime.

LiteRace's thread-local adaptive sampler is the first one listed in the table. For each thread and for each function, this sampler starts with a 100% sampling rate and then progressively reduces the sampling rate until it reaches a base sampling rate of 0.1%. To understand the utility of this adaptive back-off, a thread-local fixed sampler is evaluated. It uses a fixed 5% sampling rate per function per thread. The next two samplers are "global" versions of the two samplers that were just described. The adaptive back-off for the "global" sampler is based on the number of executions of a function, irrespective of the calling thread. This

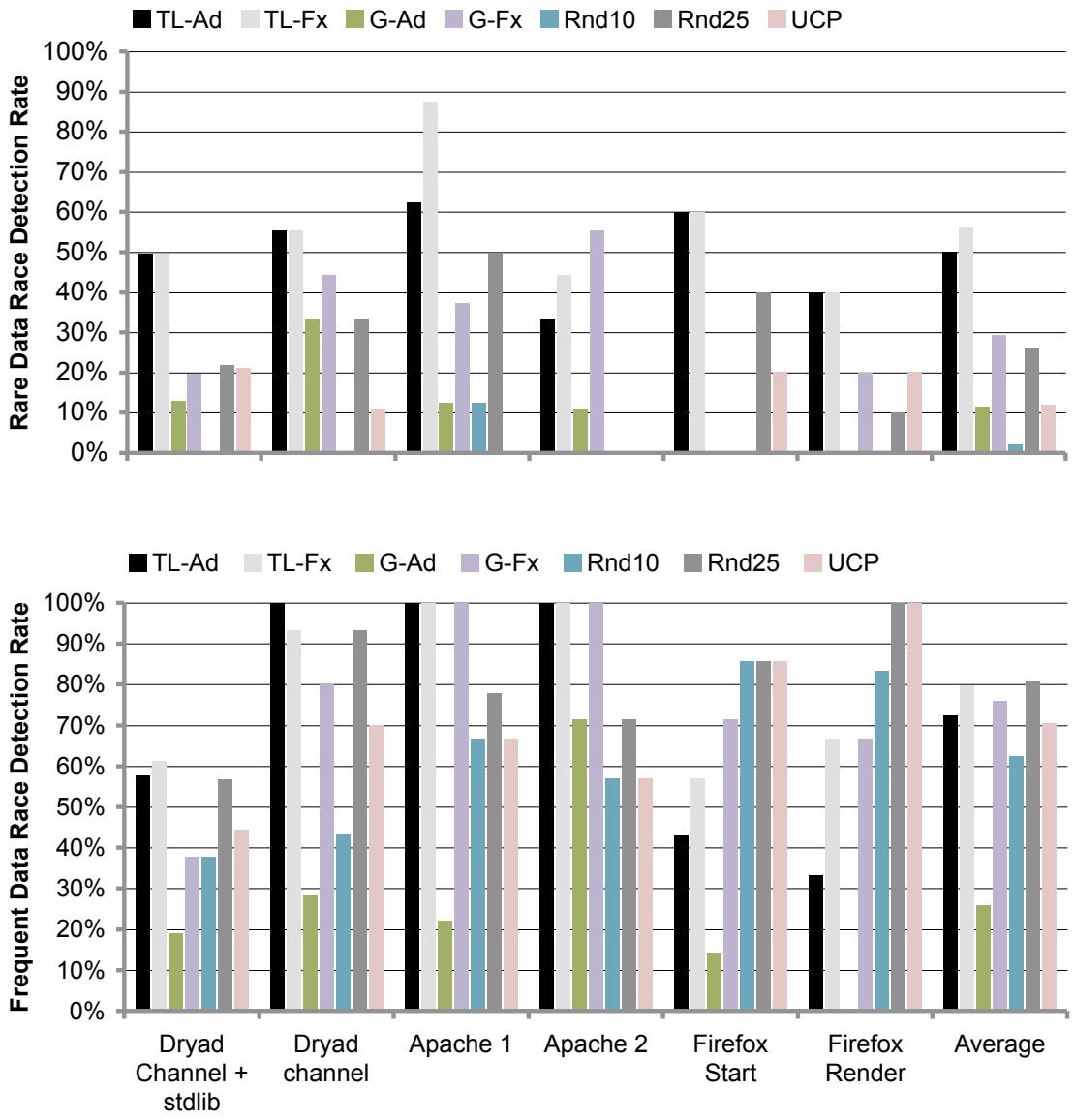

Figure 3.4: Various samplers' detection rate for rare (on the top) and frequent (on the bottom) static data races.

global adaptive sampler is similar to the one used in SWAT [HC04], except that it uses a higher sampling rate. Even with a higher rate, the experimental results show that the global samplers are not as effective as the thread-local samplers in finding data races. The four samplers mentioned thus far are "bursty". That is, when they decide to sample a function, they do so for ten consecutive executions of that function. The next two samplers are based on random sampling and are not bursty. Each function call is randomly sampled based on the chosen sampling rate (10% and 25%). The final sampler evaluates the cold-region hypothesis by logging only the "uncold" regions. That is, it logs all but the first ten calls of a function per thread.

3.3.3 Effectiveness of Samplers Comparison

In this section, the different samplers are compared and the thread-local adaptive sampler is shown to be the most effective of all the samplers evaluated. In the evaluation, the data races detected by the tool are grouped based on the pair of instructions (identified by the value of the program counter) that participate in the data race. Each group is refered to as a *static* data race. From the user's perspective, a static data race roughly corresponds to a possible synchronization error in the program. Table 3.4 shows the total number of static data races exhibited during an execution for each benchmark-input pair. The table also distinguishes between rare and frequent static data races, based on the number of times a particular static data race manifests at runtime.

To have a fair comparison, different samplers need to be evaluated on the *same* thread interleaving of a program. However, two different executions of a multi-threaded program are not guaranteed to yield the same interleaving even if the input is the same. To compare the effectiveness of the various samplers

in detecting data races accurately, a modified version of LiteRace that performs full logging was created. It logs all synchronization and all memory operations. In addition to full logging, the "dispatch check" logic for each of the evaluated samplers is executed upon function entry. The modified detector keeps track of whether or not each of the samplers would have logged a particular memory operation.

By performing data race detection on the complete log, all the data races that happened during the program's execution are found. Data race detection is then performed on the subset of the memory operations that a particular sampler would have logged. Then, by comparing the results with those from the complete log, the *detection rate* (proportion of data races detected by each of the samplers) is calculated. Note, however, that the results for performance and space overhead in §3.3.4 use the unmodified version of LiteRace with only the thread-local adaptive sampler turned on.

Each application was instrumented using the modified version of LiteRace described above. The instrumented application was run three times for each benchmark. The reported detection rate for each benchmark is the average of the three runs. The results for overall data race detection rate are shown in Figure 3.3. The results are grouped by benchmarks with a bar for each sampler within each group. The weighted average effective sampling rate for each of the samplers (discussed in §3.3.2) is also shown as the last group. A sampler is effective if it has a very low effective sampling rate along with a high data race detection rate. Notice that the proposed LiteRace sampler (TL-Ad) achieves this, as it detects about 70% of all data races by sampling only 1.8% of all memory operations. The non-adaptive fixed rate thread-local sampler also detects about 72% of data races, but its effective sampling rate is 5.2% (more than 2.5x higher

than the TL-Ad sampler). Clearly, among the thread-local samplers, the adaptive sampler is better than the fixed rate sampler.

The two thread-local samplers outperform the two global samplers. Though the global adaptive sampler logs only 1.3% of memory operations (comparable to the thread-local adaptive sampler), it detects only about 22.7% of all data races (about 3x worse than TL-Ad). The global fixed rate sampler logs 10% of memory operations, and still detects only 48% of all data races.

All the four samplers based on cold-region hypothesis are better than the two random samplers. For instance, a random sampler finds only 24% of data races, but logs 9.9% of all memory operations.

Another notable result from the figure is that of the "Un-Cold Region" sampler, which logs all the memory operations except those executed in the cold-regions (§3.3.2). It detects only 32% of all data races, but logs nearly 99% of all memory operations. This result validates the cold-region hypothesis.

Table 3.4: Number of static data races found for each benchmark-input pair (median over three dynamic executions), while logging all the memory operations. These static data races are classified into rare and frequent categories. A static data race is rare, if it is detected less than 3 times per million non-stack memory instructions during any execution of the program.

Benchmarks	**# races found**	**# Rare**	**# Freq**
Dryad Channel + stdlib	19	17	2
Dryad Channel	8	3	5
Apache-1	17	8	9
Apache-2	16	9	7
Firefox Start	12	5	7
Firefox Render	16	10	6

3.3.3.1 Rare Versus Frequent Data Race Detection

The results so far demonstrate that a thread-local adaptive sampler finds about 70% of all static data races. If a static data race occurs frequently during an execution, then it is likely that many sampling strategies would find it. It is more challenging to find data races that occur rarely at run-time. To quantify this, all of the static data races that were detected (using the full, unsampled log) were classified based on the number of times that a static data race occured in an execution. Those racing instruction pairs that occurred fewer than 3 times for each million non-stack memory instructions executed are classified as rare. The rest are considered frequent. The number of rare and frequent data races for each benchmark-input pair is shown in Table 3.4 (some of the data races found could be benign). The various samplers' data race detection rates for these two categories are shown in Figure 3.4.

Most of the samplers perform well for the frequent data races. But, for infrequently occurring data races, the thread-local samplers are the clear winners. Note that the random sampler finds very few rare data races.

3.3.4 Analysis of Overhead

§3.3.3 presented results showing that the thread-local adaptive sampler performs well in detecting data races for a low sampling rate. Here the performance and log size overhead of thread-local adaptive sampler implemented in LiteRace is described. The results show that it incurs about 28% performance overhead for the benchmarks when compared to no logging, and is up to 25 times faster than an implementation that logs all the memory operations.

Apart from the benchmarks used in §3.3.3, two additional compute and syn-

Table 3.5: Performance overhead of LiteRace's thread-local adaptive sampler and full logging implementation when compared to the execution time of the uninstrumented application. Log size overhead in terms of MB/s is also shown.

Benchmarks	Baseline Exec Time	LiteRace Slowdown	Full Logging Slowdown	LiteRace Log Size (MB/s)	Full Logging Log Size (MB/s)
LKRHash	3.3s	2.4x	14.7x	154.5	1936.3
LFList	1.7s	2.1x	16.1x	92.5	751.7
Dryad+stdlib	6.7s	1x	1.8x	1.2	12.8
Dryad	6.6s	1x	1.14x	1.1	2.6
ConcRT Messaging	9.3s	1.03x	1.08x	0.7	10.6
ConcRT Explicit Scheduling	11.5s	2.4x	9.1x	4.6	109.7
Apache-1	17.0s	1.02x	1.4x	1.2	41.9
Apache-2	3.0s	1.04x	3.2x	4.0	260.7
Firefox Start	1.8s	1.44x	8.89x	7.4	107.0
Firefox Render	0.61s	1.3x	33.5x	19.8	731.1
Average	6.15s	1.47x	9.09x	28.6	396.5
Average (w/o Microbench)	7.06s	1.28x	7.51x	5.0	159.6

chronization intensive micro-benchmarks were used for the performance study. LKRHash is an efficient hash table implementation that uses a combination of lock-free techniques and high-level synchronizations. LFList is an implementation of a lock-free linked list available from [Bus]. LKRHash and LFList execute synchronization operations more frequently than the other real world benchmarks we studied. These micro-benchmarks are intended to test LiteRace's performance in the adverse circumstance of having to log many synchronization operations.

To measure the performance overhead, each of the benchmarks was run ten times for each of four different configurations. The first configuration is the baseline, uninstrumented application. Each of the remaining three configurations adds a different portion of LiteRace's instrumentation overhead: the first adds just the dispatch check, the second adds the logging of synchronization operations, and the final configuration is the complete LiteRace instrumentation including the logging of the sampled memory operations. By running the benchmarks in all of these configurations, the cost attributable to the different components of

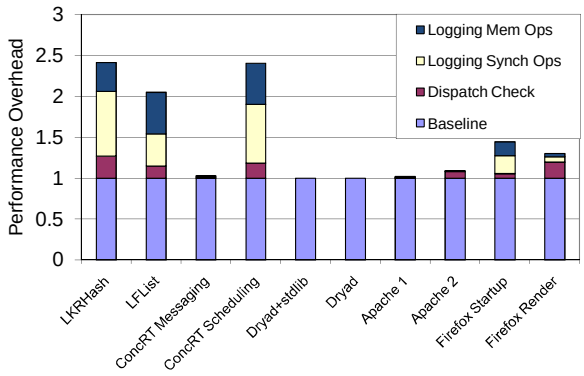

Figure 3.5: LiteRace slowdown over the uninstrumented application.

LiteRace can be measured.

Figure 3.5 shows the cost of using LiteRace on the various benchmarks and micro-benchmarks. The bottom portion of each vertical bar in Figure 3.5 represents the (normalized) time it takes to run the uninstrumented, baseline application. The overhead incurred by the various components of LiteRace are stacked on top of that. As expected, the synchronization intensive micro-benchmarks exhibit the highest overhead, between 2× and 2.5×, since all synchronization operations must be logged to avoid false positives. The ConcRT Scheduling test also has a high proportion of synchronization operations and exhibits overhead similar to the micro-benchmarks. The more realistic application benchmarks show modest performance overhead of 0% for Dryad, 2% to 4% for Apache, and 30% to 44% for Firefox.

In order to evaluate the importance of sampling memory operations in order to achieve low overhead, the performance of logging *all* the synchronization and memory accesses was measured. Unlike the LiteRace implementation, this full-logging implementation did not have the overhead for any dispatch checks or cloned code. Table 3.5 compares the slowdown caused by LiteRace to the slowdown caused by full logging. The sizes of the log files generated for these two

implementations are also shown in terms of MB/s. LiteRace performs better than full logging in all cases. The performance overhead over baseline when averaged over realistic benchmarks is 28% for the LiteRace implementation, while the full logging implementation incurs about 7.5× performance overhead.

The generated logs, as expected, are also much smaller in LiteRace. On average, LiteRace generated logs at the rate of 5.0 MB/s, whereas a full logging implementation generated about 159.6 MB/s.

3.4 LiteRace Summary

Because data races often indicate bugs in complex, multithreaded programs, dynamic data race detection can be a great boon to programmers and testers. But the high overhead of precise detectors hinders adoption and limits the number of executions on which detection can be performed, thus limiting the number of bugs which are uncovered.

By making carefully chosen sacrifices, LiteRace makes dynamic race detection for the purposes of bug finding practical. By choosing to focus effort on per-thread cold paths, LiteRace achieves a high race detection rate with a low sampling rate. By choosing to pay the cost of logging *all* synchronization operations, LiteRace avoids false positives which could cost testers and programmers precious time. The thread-local adaptive sampler manages to find nearly 70% of data races by sampling only 2% of memory accesses. This translates into low overhead (28% on average in the evaluated benchmarks) which makes running detection on a large number of executions possible.

CHAPTER 4

DRFx

The previous chapter discussed LiteRace which uses dynamic race detection to uncover potential bugs in programs. Data races can cause bugs in programs even when they are run using sequentially consistent semantics. However, as discussed previously, compilers and hardware actually provide memory models that are weaker than SC, and under these models programs cannot be intuitively reasoned about as an interleaving of the instructions from the different threads.

As discussed in Chapter 2, consensus has been building around a class of programming language memory models known as DRF0 which attempt to balance ease of programming with opportunities for compiler and hardware optimization. While these models provide programmers with a simple and strong guarantee for data-race-free programs (the observed behavior will be sequentially consistent), programmers are given much weaker guarantees, or even the possibility of completely arbitrary behavior, for programs with data races. This undermines the safety of the program as well as the ease of debugging. Furthermore, proving the correctness and safety of various compiler and hardware optimizations under some DRF0 memory models continues to be a challenge [SA08, CKS07]. The research in this chapter demonstrates how efficient, imprecise data race detection can be used to support a new memory model called DRFx that solves these problems.

4.1 Introduction to the DRFx Memory Model

The DRFx memory model uses an imprecise form of data race detection in order to provide straightforward guarantees to the programmer while still allowing most standard compiler and hardware optimizations. Despite missing some data races, the detection used provides the basis for a strong guarantee. The technique is inspired by the observation of Gharachorloo and Gibbons [GG91] 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 the 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.

4.1.1 A Compiler and Hardware Design for DRFx

Gharachorloo and Gibbons describe a hardware mechanism to detect SC violations [GG91]. 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 the 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 [GG91, CDL09].

A key contribution of DRFx is the design and implementation of a detection mechanism for SC violations that properly takes into account the effect of both compiler optimizations and hardware reorderings while remaining lightweight and efficient. The approach employs a novel form of cooperation between the compiler and the hardware. The notion of a *region*, which is a single-entry, multiple-exit

portion of a program, is introduced. The compiler partitions a program into regions, and both the compiler and the hardware may only optimize within a region. Each synchronization access must be placed in its own region, thereby preventing reorderings across such accesses. It is also required that each system call be placed in its own region, which allows DRFx to guarantee the Safety property. Otherwise, a compiler may choose regions in any manner in order to aid optimization and/or simplify runtime conflict detection. Within a region, both the compiler and hardware can perform 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, I will 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 [HM93]. As in TM systems, both software and hardware conflict detection mechanisms can be considered for supporting DRFx. A hardware detection mechanism is pursued in this implementation, since the required hardware logic is fairly simple and is similar to existing bounded hardware transactional memory (HTM) implementations such as Sun's Rock processor [DLM09]. In fact, the hardware design can be significantly simpler than that of a TM system. Unlike TM hardware, which needs complex support for versioning and checkpointing to enable rollback upon detecting a conflict, 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, an approach

discussed in §4.4 allows most of the lost optimization potential to be recovered.

4.1.2 Contributions

The research presented in this chapter makes the following contributions:

- The DRFx memory model for concurrent programming languages is defined via three simple and strong guarantees for programmers (§4.2). A set of conditions on a compiler and hardware design that are sufficient to enforce the DRFx memory model is established.

- A formalization of the DRFx memory model as well as of the compiler and hardware requirements (§4.3) is presented. A proof that these requirements are sufficient to enforce DRFx is outlined.

- A concrete compiler and hardware instantiation of the approach (§4.4) is presented. An implementation of a DRFx-compliant compiler on top of LLVM [LA04] is described, including an efficient solution for bounding region sizes so that a processor can detect conflicts using finite hardware resources.

- The performance cost of this compiler and hardware instantiation is evaluated in terms of lost optimization opportunity for programs in the Parsec and SPLASH-2 benchmark suites (§4.5). The results show that the performance overhead is on average 11% when compared to the baseline fully optimized implementation.

4.2 Overview of DRFx

This section gives an overview of how the DRFx memory model works. It first motivates and gives context by delving into more detail on some of the topics touched on in Chapter 2, including the interaction of optimizations and data races, and the impracticality of precise data race detection for the purpose of simplifying memory models. A description of the DRFx approach to targeted detection of problematic races follows.

4.2.1 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 [AH90, GLL90, MPA05]. Consider the C++ example from Figure 1.2(a) described in Chapter 1. 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 independent writes to two different memory locations. However, this reordering introduces a null dereference (and violates SC) in the interleaving shown in Figure 1.2(b).[1] The same problem can occur as a result of out-of-order execution at the hardware level.

To avoid SC violations, languages have adopted memory models based on the

[1] Although this "optimization" may seem contrived, many compiler optimizations effectively reorder accesses to shared memory. Detailed examples can be found in the next chapter (§5.2).

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

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

Figure 4.1: Correct, data-race-free version of program from Figure 1.2

DRF0 model [AH90]. Such models guarantee SC for programs that are free of data races. The data race in our example program can be eliminated by explicitly annotating the variable `init` as `atomic` (`volatile` in Java 5 and later). This annotation tells the compiler and hardware to treat all accesses to a variable as "synchronization". As such, (many) compiler and hardware reorderings are restricted across these accesses, and concurrent conflicting accesses to such variables do not constitute a data race. As a result, the revised C++ program shown in Figure 4.1 is data-race-free and its accesses cannot be reordered in a manner that violates SC.

4.2.2 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 [BA08] considers data races as errors akin to out-of-bounds array accesses and provides no semantics to racy programs. This approach requires that programmers write race-free programs in order to be able to meaningfully reason about their program's behavior. But races are a common flaw, and thus it is unacceptable to require a program be free of these bugs in order to reason about its behavior. As an example, consider the program in Figure 4.2 in which the programmer attempted to

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

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

Figure 4.2: An incorrect attempt at fixing the program from Figure 1.2.

fix the data race in Figure 1.2(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.

4.2.3 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 [AHM91, Boe09, EQT07]. Note that it is not sufficient to only detect executions that

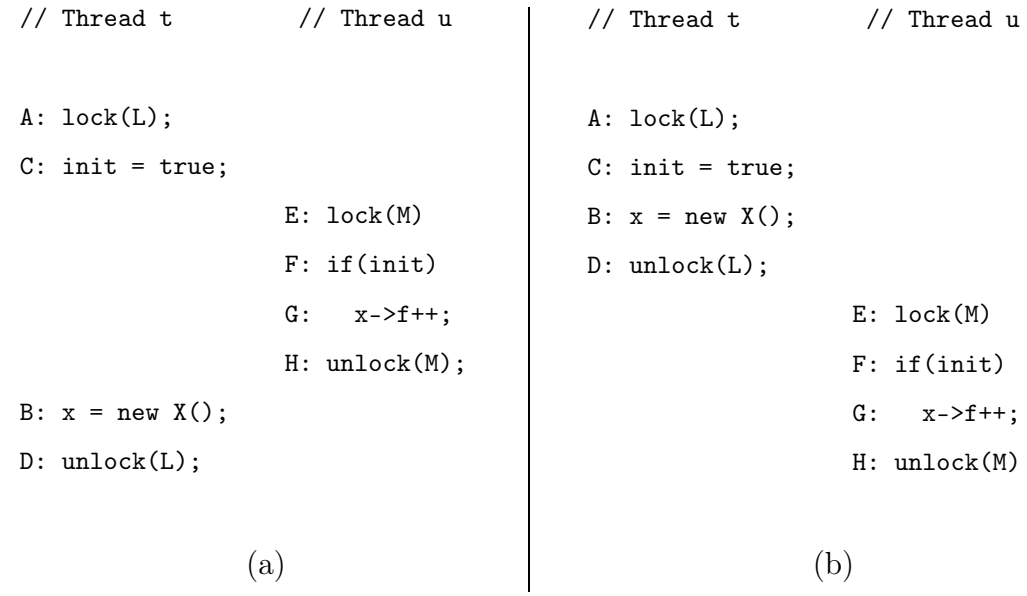

Figure 4.3: A program with a data race may or may not exhibit SC behavior at runtime. (a) Interleaving that exposes the effect of a compiler reordering. (b) Interleaving that does not.

exhibit a strictly simultaneous data race (Definition 2.2). While the existence of such an execution implies the existence of a data race in the program, other executions, which are racy only according to the more permissive Definition 2.6, can also suffer from SC violations. Figure 4.3(a) shows such an execution for the improperly synchronized code in Figure 4.2. When executing under a relaxed memory model, statements B and C can be reordered. The interleaving shown in Figure 4.3(a) suggests an execution where the racing accesses to `init` do not occur simultaneously, but non-SC behavior (null dereference upon executing statement G) can occur. The execution does have a happened-before data race by Definition 2.6.

As discussed at length in the previous chapters, precise happened-before-based data race detection is slow and thus impractical for memory model purposes. Furthermore, imprecision such as that introduced by LiteRace or prior fast detection techniques cannot provide the DRFx soundness guarantee since races resulting in

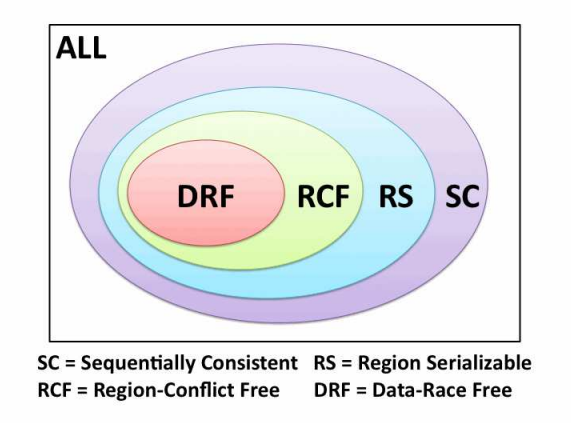

Figure 4.4: The relationships among various properties of a program execution.

violation of SC may be missed.

4.2.4 Detecting SC Violations is Enough

Although implementing DRFx requires detecting all races that may cause non-SC behavior, there are some races that do not violate SC [GG91]. Thus, full happened-before race detection, while useful for debugging, is overly strong for simply ensuring executions are SC. For example, even though the interleaving in Figure 4.3(b) contains a happened-before data race, the execution does not result in a program error. The hardware guarantees that all the memory accesses issued while holding a lock are completed before the lock is released. Since the `unlock` at D completes before the `lock` at E, the execution is sequentially consistent even though the compiler reordered the instructions B and C. Therefore, the memory model can safely allow this execution to continue. On the other hand, executions like the one in Figure 4.3(a) do in fact violate SC and should be halted with a memory model (MM) exception.

The Venn diagram in Figure 4.4 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 §4.1, all executions in `DRF` must be accepted and all executions that are not in `SC` must be terminated. However, the model allows flexibility for executions that are not in `DRF` but are in `SC`: it is acceptable to admit such executions since they are sequentially consistent, but it is also acceptable to terminate such executions since they are racy. This flexibility allows for a much more efficient detector than full-fledged race detection, as described below.

The DRFx memory model only guarantees that non-SC executions *eventually* terminate with an exception. This allows SC detection to be performed *lazily*, thereby further reducing the conflict detector's complexity and overhead. Nevertheless, the Safety property described in §4.1 guarantees that an MM exception is thrown before the effects of a non-SC execution can reach any external component via a system call.

4.2.5 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. Each synchronization operation and each system call is required to be in its own region. For instance, one possible regionization for the program in Figure 4.2 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 4.3(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 4.3(a), the conflicting regions {B,C} and {F,G} do execute concurrently, so an MM exception will be thrown.

4.2.6 From Region Conflicts to DRFx

The Venn diagram in Figure 4.4 illustrates the intuition for why the compiler and hardware co-design overviewed above 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 the 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 4.3(b). On the other hand, it is possible for an SC execution to have a concurrent region conflict and therefore trigger an MM exception. Although the execution is SC, it is nonetheless guaranteed to be racy. For example, consider again the program in Figure 4.2. 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.

4.2.7 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 4.5 describes an optimization that is disallowed by the DRFx model. Figure 4.5(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 4.5(b). The variable `sum`

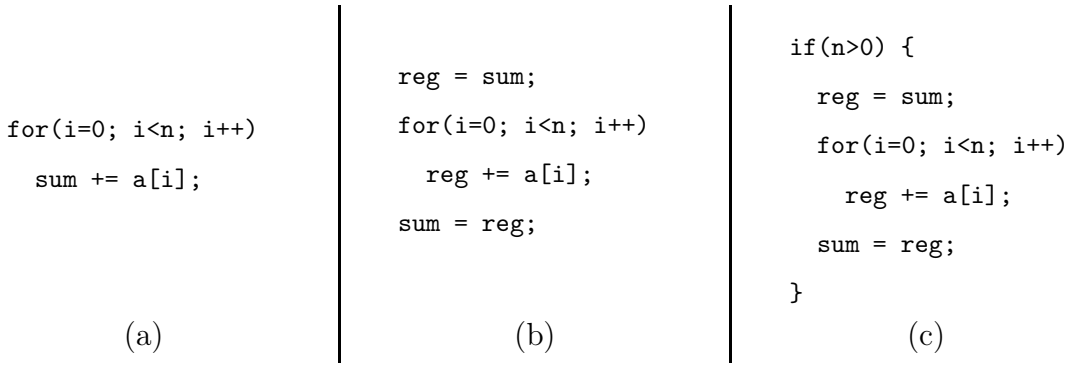

Figure 4.5: A transformation that introduces a read and a write.

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 4.5(c). The DRFx model allows the transformation with this modification, although the current compiler implementation simply disables the transformation. In spite of this, the experimental results in §4.5 indicate that the performance reduction due to lost compiler optimizations is reasonable, on average 8% on the evaluated benchmarks.

In addition to obeying the requirement above, the hardware is also responsible for detecting conflicts on concurrently executing regions. While performing conflict detection in software would avoid the need for special-purpose hardware, conflict detection in software can lead to unacceptable runtime overhead due to the need for extra computation on each memory access. On the other hand, performing conflict detection in hardware is efficient and lightweight. Sun's TM support in the Rock processor has demonstrated that conflict detection is feasible in hardware [DLM09]. DRFx hardware can actually be simpler than TM hardware, since speculation support is not needed. 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.

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

4.3 Formal Description of DRFx

This section describes the formalization of the DRFx model. Preliminary notation and definitions are introduced in §4.3.1. A formal set of requirements sufficient to establish the DRFx guarantees are broken down into the responsibilities of the compiler, and those of the execution environment, which in the implementation described in this chapter is hardware, but which could potentially be a software interpreter or some combination of hardware and software. §4.3.2 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 §4.3.3 the responsibilities of the execution environment are formalized and two important properties of a DRFx-compliant execution are established. Finally, §4.3.4 uses these results to establish the properties of the DRFx model. Full proofs are omitted here, but the interested reader can find them in prior technical reports [MSM09, SMN11].

4.3.1 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
- fence instructions (a hard fence HFENCE and a soft fence SFENCE) used only in compiled programs

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

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

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

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

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 4.1. $M_0 \rightsquigarrow M$ 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 4.2. 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.

The above definition for a racy program and execution is simply a restatement of Definitions 2.2 and 2.3 adapted to this formal setting which has a well-defined notion of an abstract execution.

4.3.2 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 4.3. 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 $2 \leq k < n$,
- $I_2, \cdots, I_{n-1} \in R$, 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 4.4. 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 the DRFx 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. Each of the fences surrounding an atomic operation must be an HFENCE. The fence preceding an END operation also must be an HFENCE.

Intuitively, the above definition of a 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 HFENCE and SFENCE 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 4.1. *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 4.2. *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 a simultaneous data race, truncate it to the earliest happened-before data race, and reorder the truncated trace while maintaining program dependencies to achieve a trace of P' with a region-sequential prefix and a suffix containing a simultaneous race. The ability to perform this reordering and achieve a region-sequential prefix relies critically on (C1) which insists that atomic accesses are in their own region. We can then use (C2) and (C3) to construct an abstract sequential execution of P exhibiting a data race from the racy execution of P'. □

Full proofs for the lemmas in this section can be found in [MSM09].

4.3.3 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 $E = (M_0, \mathcal{T}, \text{EO}, \text{RCS}, err)$. Each of the components is described below:

- M_0 is the initial machine state
- $\mathcal{T}$ is a set of individual thread traces ($\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$). Each thread trace τ_i contain instructions in the order specified by the program for thread i without gaps. We call this order TO (it totally orders instructions within a thread and is a partial order on all instructions in the program execution).

  Each thread trace is divided into *dynamic regions* (notated using metavariable ρ), with all instructions between two fence instructions in the trace belonging to one dynamic region. This is referred to as the *fence-induced partition*. We call the fence-induced partition *valid* if all atomic operations

are immediately surrounded by HFENCE instructions. Although strictly speaking TO is a relation on instructions, we will also use it to order dynamic regions within a thread trace.

- EO is a relation that specifies a partial order on memory accesses. If two operations u and v access the same memory location and at least one of them is a write, then either $u <_{\text{EO}} v$ or $v <_{\text{EO}} u$. Furthermore, no two operations that do not access the same memory location are related by EO. EO uniquely defines the write whose value each read sees (i.e., the most recent write to the same location in EO). Note that $\text{EO} \cup \text{TO}$ may contain cycles, so the relaxed orderings allowed by optimizations such as out-of-order execution and store buffers are captured by EO rather than by the thread traces.

- RCS is a map from dynamic regions to a conflict detection state in the set $\{\mathit{uncommitted}, \mathit{lagging}, \mathit{committed}\}$. Intuitively, RCS models a conflict detection mechanism which works on the fence demarcated regions and moves them through the three states as they execute, from *uncommitted*, possibly to *lagging*, and finally to *committed* when detection successfully completes with no region conflict found. The *lagging* regions will allow the model to capture a conflict detection mechanism that distinguishes between HFENCEs and SFENCEs and commits certain regions out of order.

- *err* is either $\emptyset$ or a single element of EO, $u <_{\text{EO}} v$. Intuitively, a non-empty *err* will indicate a conflicting pair of accesses in concurrently executing regions which triggers an MM exception.

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

(WF1) Each thread trace τ_i represents a valid sequential execution of thread i in P given that each read sees the value written by the (unique) closest preceding write in EO.

(WF2) Let D be the partial order that captures intra-thread data and control dependencies. EO is consistent with D and $\text{EO}_{|\text{wr}} \cup \text{D}$ 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).

(WF3) A *committed* or *lagging* region never follows an *uncommitted* region in a thread trace. That is, if there is some ρ such that $\text{RCS}(\rho) = \textit{uncommitted}$, then for all ρ' such that $\rho <_{\text{TO}} \rho'$, $\text{RCS}(\rho') = \textit{uncommitted}$.

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

(WF5) All regions preceding an HFENCE in a thread trace are *committed*. No thread trace contains an atomic access without an HFENCE immediately following it.

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

Conditions (WF3) and (WF4) establish some basic conditions that we assume for a conflict detection mechanism. Multiple *uncommitted* regions may be in-flight in a thread simultaneously. Regions may commit out of order, but when

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

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

We call a well-formed execution $E = (M_0, \mathcal{T}, \text{EO}, \text{RCS}, err)$ DRFx-compliant if it satisfies all of the following conditions, which capture a sufficient condition for conflict detection to satisfy the DRFx memory model:

(E1) Given accesses $u \in \rho_u$ and $v \in \rho_v$ to the same location from different threads, at least one of which is a write, if $u <_{\text{EO}} v$ and $\text{RCS}(\rho_u) \neq$ *uncommitted* and $\text{RCS}(\rho_v) \neq$ *uncommitted*, then there do not exist $v' \in \rho'_v$ and $u' \in \rho'_u$ such that $\rho_{u'} \leq_{\text{TO}} \rho_u$ and $\rho_v \leq_{\text{TO}} \rho_{v'}$ and $v' <_{\text{EO}} u'$. [The set of *committed* and *lagging* regions have an order consistent with EO.]

(E2) There do not exist a read $r \in \rho_r$ and a write $w \in \rho_w$ such that $\text{RCS}(\rho_r) \neq$ *uncommitted* and $\text{RCS}(\rho_w) =$ *uncommitted* and $w <_{\text{EO}} r$. [Reads in

committed and *lagging* regions do not see writes in *uncommitted* regions.]

(E3) There do not exist a read $r \in \rho_r$ and a write $w \in \rho_w$ such that $\text{RCS}(\rho_r) =$ *uncommitted* and $\text{RCS}(\rho_w) =$ *committed* and $r <_{\text{EO}} w$. [Writes from *committed* regions are visible to reads in *uncommitted* regions.]

(E4) If $err = u <_{\text{EO}} v$, then $u \in \rho_u$ and $v \in \rho_v$ conflict, u and v are from different threads, neither ρ_u or ρ_v is *committed*, and at least one of these regions is *uncommitted*.

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

A DRFx-compliant execution that has $err = \emptyset$ is called exception-free. A DRFx-compliant execution where $err \neq \emptyset$ is called exceptional.

The following lemmas establish two key results for DRFx-compliant executions.

Lemma 4.3. *Given a well-formed,* DRFx*-compliant execution E 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. This follows quickly from (E1) and (E2). Condition (E2) establishes that any value read by an instruction in $\lfloor E \rfloor$ was written by an instruction that is also in $\lfloor E \rfloor$. Furthermore, Condition (E1) establishes an order on the regions in $\lfloor E \rfloor$ that is consistent with the way that EO orders conflicting accesses within those regions. This ensures that the lifting of the EO relation to dynamic regions is acyclic, which implies that the execution is serializable w.r.t. the regions. □

Lemma 4.4. *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 Lemma 4.3 we know that the execution has a region-serializable prefix. We then use this to construct an abstract sequential execution of the prefix. Because the execution is exceptional, condition (E4) guarantees that we have conflicting accesses neither of which is contained in a *committed* region, and at least one of which is from an *uncommitted* region. We can extend the execution of the prefix to an execution demonstrating a happened-before data race. Essentially, for a program with valid fence-induced thread partitions, a happened-before relation between operations on different threads implies the existence of an HFENCE following one operation on its thread and preceding the other on its thread. Since neither of the conflicting accesses is from a *committed* region, and condition (WF5) requires regions preceding an HFENCE to be *committed*, we know the accesses cannot be related by happened-before. [2] □

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

Full proofs for the previous two lemmas can be found in [SMN11]. Rather than starting with the conditions for well-formed, DRFx-compliant execution, the proofs in the cited techical report are done in the context of the particular architectural design described in §4.4.2. Conditions (E1) through (E4) capture the supporting lemmas from the technical report that are used to establish the results above. [3]

4.3.4 DRFx Guarantees

Putting together the lemmas from Sections 4.3.2 and 4.3.3, we can prove the following theorem, which ensures that a DRFx-compliant compiler along with a DRFx-compliant execution environment enforce the DRF and Soundness properties. We call an execution *complete* if either it is exceptional (contains a non-null *err* component), or all the thread traces in the execution terminate in an END operation.

Theorem 4.1. *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

[3] Note that an earlier technical report [MSM09] establishes similar results under a different set of conditions that were too restrictive for the eventual hardware design.

operands from memory into registers, perform appropriate arithmetic operations on the registers, and then store results back to memory. Any optimizations are then applied after this canonical compiler is run. We can extend the results to the high-level language simply by requiring that the compiler choose a region partition that does not split up instructions that came from the same high-level source language expression or statement.

The definition of a DRFx-compliant execution and Lemma 4.3 establish that all DRFx-compliant executions are region-serializable up to the latest *committed* region in each thread. Combining this fact with Lemma 4.1, we can see that, restricted to *committed* and *lagging* regions, a DRFx-compliant execution is SC with respect to the original source program. Note that an HFENCE operation cannot execute until all previous regions in its thread are *committed* (condition (WF5)). Therefore, requiring that system calls are preceded by HFENCE instructions and only use thread-local data ensures that the behavior they exhibit would have been achievable in an SC execution of the original program.[4] This establishes the Safety property of the DRFx model.

4.4 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 one concrete approach is described. It is evaluated in the next section. The approach is based on two key ideas crucial for a simple hardware design:

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

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

- **Soft fences:** When splitting regions to guarantee boundedness, the compiler inserts a *soft* fence. Soft fences are distinguished from the fences used to demarcate synchronization operations and system calls which are called *hard* fences. While 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, it is in fact safe for the hardware to reorder instructions across soft fences whenever hardware resources are available, essentially erasing any hardware performance penalty due to the use of bounded-size regions.

4.4.1 Compiler Design

A DRFx-compliant compiler was built by modifying the LLVM compiler [LA04]. As specified by the requirements (C1) through (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.

4.4.1.1 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 and the compiler is disallowed from optimizing across them. To guarantee SC for race-free programs, the compiler must insert a hard fence before and after each synchronization access. On some architectures, the synchronization access itself can be translated to an instruction that has hard-fence semantics (e.g., the atomic `xchg` instruction in AMD64 and Intel64 [BA08]), obviating the need for additional fence instructions. In the current implementation, the compiler treats all calls to the `pthread` library and lock-prefixed memory operations as "atomic" accesses. In addition, since the LLVM compiler does not support the `atomic` keyword proposed in the new C++ standard, all `volatile` variables are treated as atomic. All other memory operations are treated as data accesses.

To guarantee DRFx's Safety property, a DRFx-compliant compiler should also insert hard fences for each system call invocation, one before entering the kernel mode and another after exiting the kernel mode. Any state that could be read by the system call should first be copied into a thread-local data structure before the first hard fence is executed. This approach ensures that the external system can observe only portions of the execution state that are reachable in some SC execution. Transforming system calls in this way is not implemented in the compiler used for the experiments in §4.5.

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

4.4.1.2 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. Soft fences are inserted using a newly created intrinsic instruction in LLVM that is compiled to a special x86 no-op instruction which can be recognized by the DRFx hardware simulator as a soft fence. The compiler employs a simple and conservative static analysis to bound the number of memory operations in a region. While overly small regions do limit the scope of compiler optimizations, experiments show that the performance loss due to this limitation is about 1.7% on average [MSM10]. After inserting all the hard fences described earlier, the compiler performs function inlining. Soft fences are the inserted in the inlined code. A soft fence is conservatively inserted before each function call and return, and before each loop back-edge. Finally, the compiler inserts additional soft fences in a function body as necessary to bound region sizes. The compiler performs a conservative static analysis to ensure that no region contains more than R memory operations, thereby bounding the number of bytes that can be accessed by any region. The constant R is determined based on the size of hardware buffers provisioned for conflict detection.

The above algorithm prevents compiler optimizations across loop iterations, since a soft fence is inserted at each back-edge. However, it would be possible to apply a transformation similar to loop tiling [Wol89] which would have the effect of placing a soft fence only once every R/L iterations, where L is the maximum number of memory operations in a single loop iteration. Restructuring loops in this way would allow the compiler to safely perform compiler optimizations across each block of R/L iterations.

4.4.1.3 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 since they can cause false conflicts. As such, in the current implementation, all speculative optimizations in LLVM are explicitly disabled.[5] Note, however, that there are several useful speculative optimizations that have simple variants that would be allowed by the DRFx 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 [TBB01] in order to hide the memory latency of reads. Also, as shown earlier in Figure 4.5, loop-invariant code motion is allowed by the DRFx model, as long as the hoisted reads and writes are guarded to ensure that the loop body will be executed at least once.

4.4.2 Hardware Design and Implementation

This section discusses the proposed DRFx processor architecture. A lazy conflict detection scheme using bloom filter signatures is described, as well as several optimizations that allow efficient execution in spite of the small, bounded regions created by the DRFx compiler.

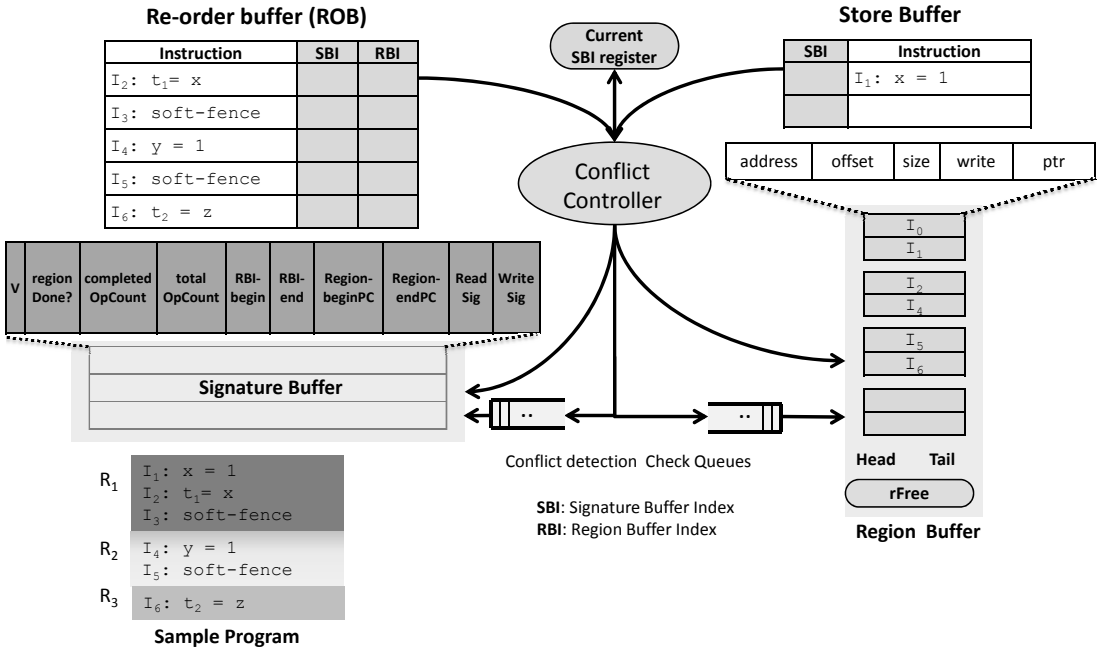

Figure 4.6: Architecture support for DRFx (shown in gray).

4.4.2.1 Overview

To satisfy DRFx 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 (§4.2.6). Figure 4.6 presents an overview of a DRFx 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.

Rollback is a necessary requirement of hardware transactional memory systems. As such, they can easily tolerate false positives in their conflict detection mechanism by simply rolling back and re-executing. This allows them to use

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

cache-line granularity conflict detection which may report false races. DRFx, on the other hand, does not require a rollback mechanism. But, because it terminates an execution upon detecting a race, false race reports cannot be tolerated. As such, DRFx performs byte-level conflict detection. Performing precise, *eager* byte-level conflict detection complicates the coherence protocol and cache architecture [LCS10]. For instance, such a scheme would require the hardware to maintain byte-level access state for every cache block, maintain the access state even after a cache block migrates from one processor to another, and clear the access state in remote processors when a region commits.

Instead, the DRFx hardware employs lazy conflict detection [HCW04]. 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 load *completes* its execution when it *commits* from the reorder buffer, while a store *completes* its execution when it *retires* 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. The communication and conflict check overhead is reduced by using bloom filter signatures to represent sets of addresses [CTT06]. 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 the DRFx compiler. The static analysis used by the DRFx 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. This cost is reduced by coalescing adjacent regions separated by a soft fence into a single region at runtime when there is sufficient space available in the region buffer. Supporting this optimization requires using a region buffer somewhat larger than the maximum possible region-size guaranteed by the compiler.

When executing a hard fence, the DRFx hardware stalls the execution of all future memory accesses until all accesses preceding the fence have completed. This helps guarantee correct behavior of synchronization operations and ensures that any conflicts that are detected indeed correspond to a data race. But it also prevents full utilization of processor resources since instruction and memory level parallelism cannot be exploited across the fence. If the more frequently occurring soft fences behaved the same as hard fences, these lost opportunities to exploit parallelism would result in significant performance overhead. Fortunately, this is unnecessary since soft fences do not indicate the presence of synchronization. In fact, memory accesses from a region can be allowed to execute even if earlier regions that end in soft fences have not committed. In addition, regions separated by a soft fence can be committed out of order. The formal proofs outlined in Section 4.3 admit these optimizations and establish that the DRFx runtime requirements are still satisfied.

4.4.2.2 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, bloom filter signatures [CTT06] can be used. 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 the out-of-order execution optimizations discussed later in Section 4.4.2.5). 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. Experimental results show that the dynamic region size is relatively small (36 instructions, on average). But, since many regions can be in flight in a processor at once, the signature may be compared with many remote regions, increasing the probability of getting a false conflict. To address this problem, large signatures (1024 bits) are used, but they are compressed before transmission to reduce communication overhead. Because many regions have small access sets, their signatures are effectively compressed using a simple, efficient run-length encoding scheme. This strategy resulted in very high compression ratios which significantly reduced communication overhead.

Note that the 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.

4.4.2.3 Concurrent Region Conflict Check and Region Execution

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. Conflicts involving a memory access following a mispredicted branch were very infrequent in the experiments, therefore the cost of delaying the 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, signatures for its region are not updated. This could result in additional false positives, but the performance impact is unlikely to be important.

4.4.2.4 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. But a processor can dynamically 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. The design guarantees this by using

a region buffer that is of size `T + B`, where `B` is the compiler specified bound for a soft-fenced region, and `T` is the threshold used by a processor to determine when to ignore a soft fence. Too high a value for the threshold `T` would result in large regions at runtime, which might negatively impact performance, because the probability of aliases in signatures increase. Also, it could undermine out-of-order commit optimization.

4.4.2.5 Out-of-Order Execution and Commit of Regions

Two important restrictions that need to be obeyed for hard fences can be relaxed for soft fences, which allows DRFx hardware to attain performance close to DRF0.

First, out-of-order execution of soft-fenced regions is allowed. 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. 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, this memory ordering can be relaxed for soft fences, allowing multiple regions that are not committed to be in-flight simultaneously. For example, in Figure 4.7, $\mathtt{I}_7$ can be allowed to execute even if regions $\mathtt{R}_0$ and $\mathtt{R}_1$ have pending memory accesses in the ROB or the store buffer. If there is a pending store in a previous region (e.g., $\mathtt{I}_1$), its value can be forwarded to a load in a later region (e.g., $\mathtt{I}_7$).

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, reordering accesses across soft fences will not cause any access to be reordered across a synchronization

operation. As such, any conflict that is detected as a result of this reordering still implies the presence of a data race.

Second, 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. In this scenario, 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. The not yet committed, prior regions correspond to the *lagging* regions in the formalism described in §4.3.3. In order to satisfy conditions (E1) and (E2) for *lagging* regions, addresses for the uncommitted, previous regions must be included in the conflict check message for the later region.

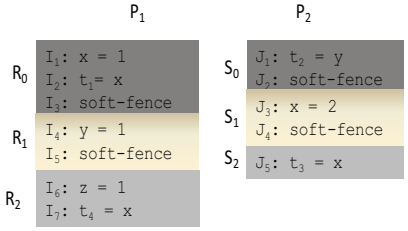

Figure 4.7: An Example Binary Compiled Using DRFx Compiler.

For example, in Figure 4.7, say region $\mathtt{R_0}$ is waiting for its store $\mathtt{I_1}$ to be retired from the store buffer. In the meantime, $\mathtt{I_4}$ has completed and has retired from the store buffer. Now $\mathtt{R_1}$ is ready to commit. The processor can perform conflict checking for $\mathtt{R_1}$ (including the addresses for any uncommitted, prior regions), and if no conflict is detected, commit by deleting its entries from the region and

signature buffers (but leaving the entries for uncommitted, prior regions). This optimization can be intuitively understood by observing that even if a write from R_0 lingering in the store buffer eventually causes a conflict with another access that has not yet had its address resolved, the successful conflict check of the addresses in R_1 and R_0 at the time R_1 commits establishes a global order of all committed and lagging regions in the system at that point. This guarantees SC behavior up to the latest committed region in each thread.

4.5 Performance Evaluation

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

The baseline compiler is the LLVM [LA04] compiler with all optimizations enabled (similar to compiling with the `-O3` flag in `gcc`) and with fences inserted before and after each call to a synchronization function and each access to a volatile variable.[6] The DRFx compiler is the implementation described in the previous section: hard fences are inserted before each call to a synchronization function and each access to a volatile variable, optimizations that perform speculative reads or writes are disabled, and soft fences are inserted to conservatively bound region size to 512 memory accesses.

Both the baseline and DRFx architectures are simulated using a cycle-accurate,

[6]The unmodified LLVM compiler using its x86 backend targets hardware obeying the TSO memory model. The baseline simulated architecture uses a weaker memory model which permits additional reorderings not allowed by TSO. As such, we insert the additional fences around synchronization accesses to ensure that the program behaves correctly on the weaker model. The benchmarks run slightly faster in this baseline, DRF0 configuration than on a simulated TSO architecture running code compiled with unmodified LLVM.

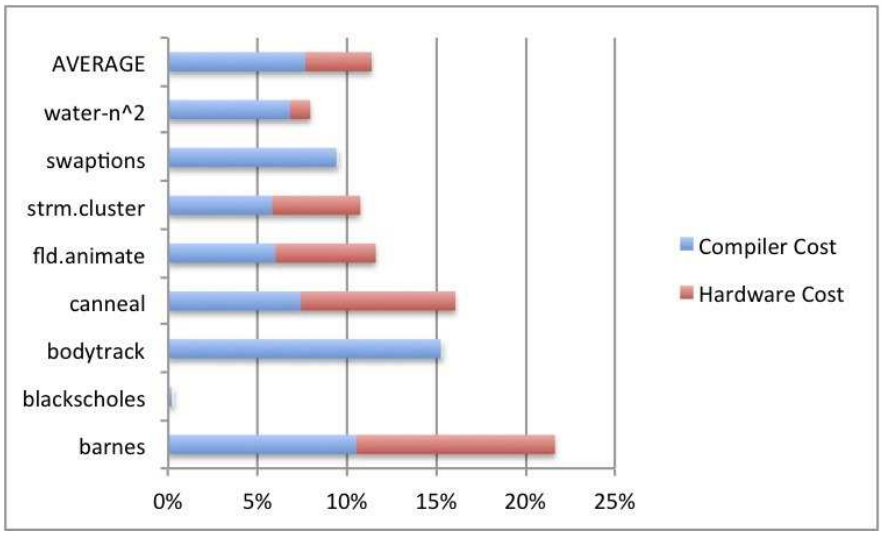

Figure 4.8: Slowdown of benchmark programs run under the DRFx model compared to a baseline DRF0 model, broken down in terms of cost of lost compiler optimization and cost of hardware race detection.

execution driven, Simics based x86_64 simulator called FeS2 [FeS]. The baseline architecture is a 4-core chip multiprocessor operating at 2GHz. It allows both loads and stores to execute out of order between fences. The DRFx architecture adds support for soft fences and conflict detection as described in the previous section, using a region buffer of size 512 (compiler bound) + 32 (to support region coalescing).

Performance is measured over a subset of the PARSEC [BKS08] and SPLASH-2 [WOT95] benchmarks. All of these benchmarks are run to completion. For PARSEC benchmarks (blackscholes, bodytrack, canneal, facesim, streamcluster, swaptions), the `sim-medium` input set was used (except for streamcluster, which used the `sim-small` input). For SPLASH-2 applications (barnes, water-n^2) the default inputs were used.

The slowdown of the DRFx configuration (compiler and hardware) over the baseline configuration is shown in Figure 4.8. The bar is broken up to display the cost that can be attributed to compiler optimizations that were not able to be

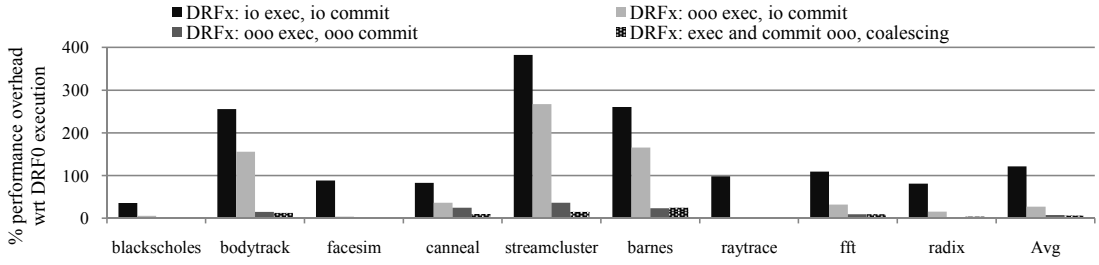

Figure 4.9: Effectiveness of Region Coalescing, and Out-Of-Order Region Execution and Commit Optimizations.

performed and the cost of hardware detection to support DRFx. The approximate cost of lost compiler optimizations was calculated by compiling a benchmark program using the DRFx compiler, converting soft fences to no-ops, and running the resulting binary on the baseline DRF0 hardware simulator. On average, applications suffer only an 11% slowdown, 8% of which comes from lost optimization opportunities in the compiler. As mentioned in the previous section, this DRFx compiler implementation is quite conservative and much of this performance could likely be recovered if loop optimizations were updated to be DRFx-compliant.

Figure 4.9 demonstrates the importance of distinguishing soft fences and implementing the optimizations described in the previous section. When soft fences are treated like hard fences, the benchmarks slow by more than 2× on average. Enabling out-of-order execution and region commit for soft-fence-bounded region and region coalescing reduces this drastically to an 11% overhead.

4.6 Conclusion

The DRFx memory model for concurrent programming languages gives programmers simple, strong guarantees for all programs. 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 weak or no 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 and enables programmers to easily reason about *all* programs using the intuitive SC semantics. Furthermore, the minor restrictions DRFx places on compiler optimizations are straightforward, allowing compiler writers to easily establish the correctness of their optimizations.

DRFx capitalizes on the fact that sequentially-valid compiler optimizations preserve SC as long as they do not interact with concurrent accesses on other threads. Since performing precise data race detection is impractically slow in software and complex in hardware, DRFx allows the compiler to specify code regions in which optimizations were performed. The hardware can then efficiently target data race detection only at regions of code that execute concurrently. This allows DRFx-compliant compiler and hardware to cooperate, terminating executions of racy programs that may violate SC. The formal development establishes a set of requirements for the compiler and the hardware that are sufficient to obey the DRFx model. The implementation and evaluation indicate that a high-performance implementation of DRFx is possible.

CHAPTER 5

An SC-preserving Compiler

Like the previous chapter which described DRFx, this chapter describes an approach to simplifying the memory model exposed to the programmer. The approach is in some ways similar to DRFx and in some ways starkly different. While DRFx encompasses both the hardware and programming language level memory models, the approach described in this chapter divorces the two. The focus is on avoiding SC-violating optimizations in the compiler while relying on existing hardware techniques that separately provide a strong memory model. This essentially allows us to pass the hardware-level memory model through to the programming language level unchanged. As in DRFx, programmers are given guarantees for both racy programs and data-race-free programs. Also like DRFx, a form of hardware race detection and cooperation between the compiler and hardware is used. But in this approach, it is primarily used as a performance optimization as opposed to being central to correctness. The key in this approach is restricting the compiler to reorder memory accesses only when it can guarantee that those accesses are to thread-local variables, which is accomplished using a simple, conservative, modular analysis. This can be thought of as statically finding slices of code which cannot contain racing accesses and thus can be safely optimized.

5.1 Introduction

Part of the motivation of DRFx, and indeed of the DRF0 memory models before it, is the commonly accepted assumption that programming languages must relax the SC semantics of programs in order to allow effective compiler optimizations. The research presented in this chapter challenges that assumption by demonstrating an optimizing compiler that retains most of the performance of the generated code while preserving the SC semantics. A compiler is said to be *SC-preserving* if every SC behavior of a generated binary is guaranteed to be an SC behavior of the source program.

Starting from LLVM [LA04], a state-of-the-art C/C++ compiler, the SC-preserving compiler was built by modifying each of the optimization passes to conservatively disallow transformations that might violate SC. Experimental results (Section 5.3) indicate that the resulting SC-preserving compiler incurs only 3.8% performance overhead on average over the original LLVM compiler with all optimizations enabled on a set of 30 programs from the SPLASH-2 [WOT95], PARSEC [BKS08], and SPEC CINT2006 (integer component of SPEC CPU2006 [Hen06]) benchmark suites. Moreover, the maximum overhead incurred by any of these benchmarks is just over 34%.

5.1.1 An Optimizing SC-Preserving Compiler

The observation that enables this approach is that a large class of optimizations crucial for performance are either already SC-preserving or can be modified to preserve SC while retaining much of their effectiveness. Several common optimizations, including procedure inlining, loop unrolling, and control-flow simplification, do not change the order of memory operations and are therefore

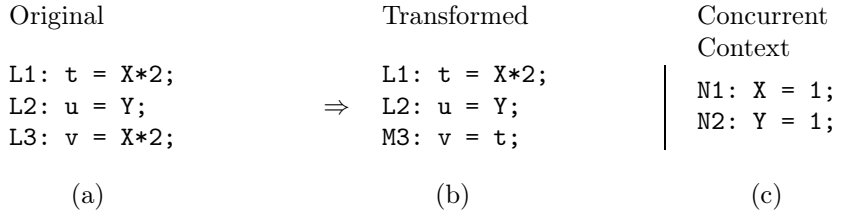

Figure 5.1: A compiler transformation from program (a) into (b) that eliminates the common subexpression `X*2`. In the presence of a concurrently running thread (c) and an initial state where all variables are zero, (b) can observe a state `u == 1 && v == 0`, which is not visible in (a). Lowercase variables denote local temporaries, while uppercase variables are potentially shared.

naturally SC-preserving. Other common optimizations, such as common subexpression elimination (CSE) and loop-invariant code motion, can have the effect of reordering memory operations. However, these optimizations can still be performed on accesses to thread-local variables and compiler-generated temporary variables. The analysis required to distinguish such variables is simple, modular, and is already implemented by modern compilers such as LLVM. Furthermore, transformations involving a single shared variable are also SC-preserving under special cases (Section 5.2).

Consider the instance of CSE in Figure 5.1, where the compiler eliminates the subexpression `X*2`. By reusing the value of `X` read at `L1` in `L3`, this transformation effectively reorders the second access to `X` with the access to `Y` at `L2`. While invisible to sequential programs, this reordering can introduce non-SC behaviors in a concurrent program, as shown in Figure 5.1. However, an SC-preserving compiler can still perform this transformation as long as at least one of `X` and `Y` is known to be thread-local. If `X` is thread-local, then its value does not change between `L1` and `L3` and so the transformation is SC-preserving. On the other hand, if `Y` is thread-local then any SC execution of the transformed program can be shown to be equivalent to an SC execution of the original program in which instructions `L1` to `L3` execute without being interleaved with instructions from

other threads. By carefully enabling transformations only when they are SC-preserving, a compiler is able to achieve performance comparable to a traditional optimizing compiler while retaining the strong SC semantics.

5.1.2 Providing End-to-End Programmer Guarantees

Providing end-to-end SC semantics to the programmer requires executing the output of an SC-preserving compiler on SC hardware. The empirical results in this chapter complement recent architecture research [GGH91, RPA97, Hil98, GF02, CTM07, BMW09] that demonstrates the feasibility of efficient SC hardware. The basic idea behind these proposals is to speculatively reorder memory operations and recover in the rare case that these reorderings can become visible to other processors. While such speculation support necessarily increases hardware complexity, hopefully this work on an SC-preserving compiler increases the incentives for building SC hardware, since in combination they enable end-to-end SC semantics for programmers at a reasonable cost.

Even in the absence of SC hardware, the techniques described in this chapter can be used to provide strong semantics to the programmer. For instance, when compiling to x86 hardware, which supports the relatively-strong total store order (TSO) memory model [OSS09], a compiler that preserves TSO behavior guarantees TSO semantics at the programming language level even for racy programs. Data-race-free programs continue to enjoy SC semantics in this scenario. The result is a language-level memory model that is stronger and simpler than the current memory-model proposals for C++ [BA08, BOS11] and Java [MPA05], given that programs may contain data races.

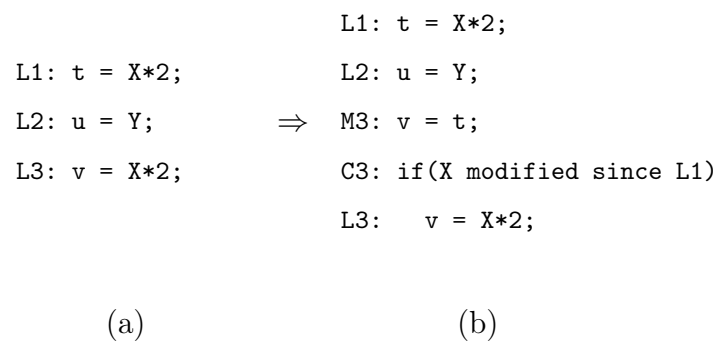

Figure 5.2: Performing common subexpression elimination while guaranteeing SC. The interference check at `C3` ensures that the value of `X` has not changed since last read at `L1`. This allows the compiler to reuse the value of `X*2` computed in `L1` without violating SC.

5.1.3 Speculative Optimization For SC-Preservation

While the cost of an SC-preserving compiler is much less than previously assumed, one possible concern is that some applications might be unwilling to pay this cost, however small. Nevertheless, one should exhaust possible avenues for improving the performance of SC-preservation, such as more sophisticated static and dynamic analyses, before exposing a relaxed program semantics to the programmer.

In this vein, consider the fact that many of the disabled optimizations responsible for lost performance in the SC-preserving compiler involve an *eager load*. For instance, the elimination of the expression `X*2` in Figure 5.1 can be considered as performing the load of variable `X` eagerly at line `L1` instead of at `L3`. Other eager-load optimizations include constant propagation, copy propagation, partial-redundancy elimination, global value numbering, and common cases of loop-invariant code motion. Experiments show that fully enabling these eager-load optimizations in the SC-preserving compiler reduces the maximum slowdown of any benchmark from 34% to 6.5%.

§5.4 explains how to enable eager-load optimizations without violating SC.

Compiler-inserted *interference checks* are used to dynamically ensure the correctness of optimizations that cannot be statically validated as SC-preserving. Figure 5.2 demonstrates this idea. The figure shows the code from Figure 5.1(a) and its transformation with an interference check. For the CSE optimization to be sequentially valid, the compiler already ensures that the variable `X` is not modified by instructions between `L1` and `L3`. The interference check lifts this correctness requirement to concurrent programs by ensuring that no other thread has modified `X` since last read at `L1`. If the check succeeds, the program can safely reuse the earlier computation of `X*2`; if not, the program reverts to the unoptimized code.

These interference checks are a form of *targeted, compiler-directed* data race detection. They indicate that races must be detected for a particular location during execution of a particular code region. Furthermore, the detection mechanism can be conservative (that is, it can report a false positive), without causing problems since the compiler is required to insert code to recover in case of a race. This fine-grained detection is quite different from the detection required by DRFx, which cannot report false races and which must perform detection for all memory accesses.

The interference checks are inspired by a common hardware speculation mechanism [GGH91] that is used to safely strengthen hardware memory models. This mechanism allows a processor to track cache-coherence messages to conservatively detect when a particular memory location may have been modified by another processor. An extension of this speculation mechanism can be used to discharge interference checks efficiently. §5.5 describes a simple interface for exposing this capability to the compiler, based on the Itanium architecture's design of a similar feature [Ita02]. A hardware simulator supporting the speculation mechanism and

a) redundant load: `t=X; u=X;` ⇒ `t=X; u=t;`

b) forwarded load: `X=t; u=X;` ⇒ `X=t; u=t;`

c) dead store: `X=t; X=u;` ⇒ `X=u;`

d) redundant store: `t=X; X=t;` ⇒ `t=X;`

Figure 5.3: SC-preserving transformations

a simulation study on 15 parallel programs from the SPLASH-2 and PARSEC benchmarks are described in §5.6. By incorporating interference checks into a single optimization pass, the average performance overhead of the SC-preserving compiler on simulated TSO hardware is reduced from 3.4% to 2.2% and the maximum overhead is reduced from 23% to 17%.

5.2 Compiler Optimizations as Memory Reorderings

In this section, compiler optimizations are classified based on how they affect the memory reorderings of the program [AG96, SA08].

5.2.1 SC-Preserving Transformations

Informally, the (SC) behaviors of a program can be represented as a set of interleavings of the individual memory operations of program threads that respect the per-thread program order. A compiler transformation is SC-preserving if every behavior of the transformed program is a behavior of the original program. Note that it is acceptable for a compiler transformation to reduce the set of behaviors.

Transformations involving thread-local variables and compiler-generated temporaries are always SC-preserving. Furthermore, some transformations involving a single shared variable are SC-preserving [SA08]. For example, if a program

performs two consecutive loads of the same variable, as in Figure 5.3(a), the compiler can remove the second load. This transformation preserves SC as any execution of the transformed program can be emulated by an interleaving of the original program where no other thread executes between the two loads. On the other hand, this transformation reduces the set of behaviors, as the behavior in which the two loads see different values is not possible after the transformation.

Similar reasoning can show that the other transformations shown in Figure 5.3 are also SC-preserving. Further, a compiler can perform these transformations even when the two accesses on the left-hand side in Figure 5.3 are separated by local accesses, since those accesses are invisible to other threads.

5.2.2 Ordering Relaxations

Optimizations that are not SC-preserving change the order of memory accesses performed by one thread in a manner that can become visible to other threads. We characterize these optimizations based on relaxations of the following ordering constraints among loads and stores that they induce: `L` $\rightarrow$ `L`, `S` $\rightarrow$ `L`, `S` $\rightarrow$ `S`, and `L` $\rightarrow$ `S`.

Consider the CSE example in Figure 5.1(a). This optimization involves relaxing the `L` $\rightarrow$ `L` constraint between the loads at `L2` and `L3`, moving the latter to be performed right after the first load of `X` at `L1`, and eliminating it using the transformation in Figure 5.3(a). If the example contained a store, instead of a load, at `L2`, then performing CSE would have involved an `S` $\rightarrow$ `L` relaxation. We classify an optimization as an *eager load* if it only involves `L` $\rightarrow$ `L` and `S` $\rightarrow$ `L` relaxations, as these optimizations involves performing a load earlier than it would have been performed before the transformation.

Another example of an eager load optimization is constant/copy propagation

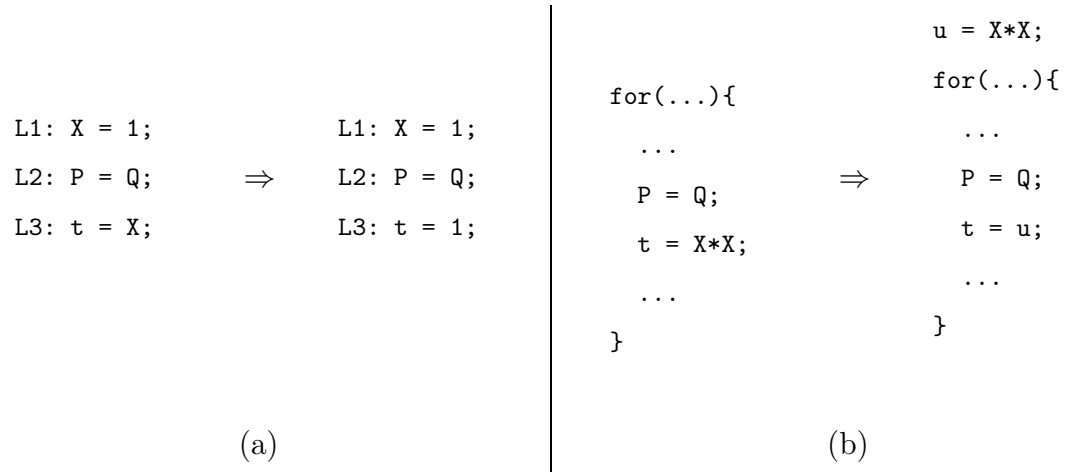

Figure 5.4: Examples of eager-load optimizations include constant/copy propagation (a) and loop-invariant code motion (b). Both involve relaxing the L → L and S → L ordering constraints.

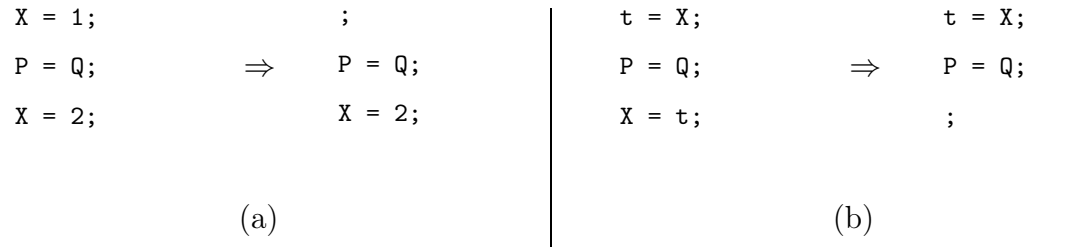

Figure 5.5: (a) Dead store elimination involves relaxing the S → S and S → L constraints. (b) Redundant store elimination involves relaxing the L → S and S → S constraints.

as shown in Figure 5.4(a). In this example, the transformation involves moving the load of X to immediately after the store of X (which requires L → L and S → L relaxation with respect to the P and Q accesses) and then applying the transformation in Figure 5.3(b). The loop-invariant code motion example in Figure 5.4(b) involves eagerly performing the (possibly unbounded number of) loads of X within the loop once before the loop. This also requires relaxing L → L and S → L ordering constraints due to the store and load to shared variables P and Q respectively.

Figure 5.5 shows examples of optimizations that are *not* eager loads. The dead-store elimination example in Figure 5.5(a) involves relaxing the S → S and

S → L constraints by delaying the first store and then applying the SC-preserving step of combining the adjacent stores as in Figure 5.3(c). Figure 5.5(b) shows an example of a redundant store elimination that involves eagerly performing the store of X by relaxing the L → S and S → S ordering constraints and then applying the transformation in Figure 5.3(d).

5.3 An SC-Preserving Modification to LLVM

This section describes the design and implementation of the optimizing SC-preserving compiler on top of LLVM and evaluates the compiler's effectiveness in terms of performance of the generated code versus that of the baseline LLVM compiler.

5.3.1 Design

As described in the previous section, we can characterize each compiler optimization's potential for SC violations in terms of how it reorders memory accesses. In order to build the SC-preserving compiler, each transformation pass performed by LLVM was examined to determine whether or not it could potentially reorder accesses to shared memory. The passes were further categorized based on what types of accesses might be reordered.

Perhaps surprisingly, many of LLVM's passes do not relax the order of memory operations at all and these SC-preserving passes can be left unmodified. These passes include sparse conditional constant propagation, dead argument elimination, control-flow graph simplification, procedure inlining, scalar replication, allocation of function-local variables to virtual registers, correlated value propagation, tail-call elimination, arithmetic re-association, loop simplification,

Table 5.1: This table lists the passes performed by a standard LLVM compilation for an x86 target that have the potential to reorder accesses to shared memory. The table indicates which memory orderings may be relaxed and whether the SC compiler disables the pass entirely or modifies it to avoid reordering.

Short Nm	**Description**	L → L	L → S	S → L	S → S	**SC Vers.**
LLVM IR Optimization Passes						
inst-combine	Performs many simplifications including algebraic simplification, simple constant folding and dead code elimination, code sinking, reordering of operands to expose CSE opportunities, limited forms of store-to-load forwarding, limited forms of dead store elimination, and more.	yes	no	yes	no	modified
arg-promotion	Promotes by-reference parameters that are only read into by-value parameters; by-value struct types may be changed to pass component scalars instead.	yes	no	yes	no	disabled
jump-threading	Recognizes correlated branch conditions and threads code directly from one block to the correlated successor rather than executing a conditional branch. While this threading in itself would not reorder memory accesses, this pass performs some partially redundant load elimination to enable further jump threading, and that may have the effect of performing an eager load.	yes	no	yes	no	modified
licm	Performs loop-invariant code motion and register promotion.	yes	yes	yes	yes	modified
gvn	The global value numbering pass performs transformations akin to common subexpression elimination, redundant and partially redundant load elimination, and store-to-load forwarding.	yes	no	yes	no	modified
memcpy-opt	Performs several optimizations related to memset, memcpy, and memmov calls. Individual stores may be replaced by a single memset. This can cause observable reordering of store operations (e.g. `A[0]=-1; A[2]=-1; A[1]=-1` becomes `memset(A,-1,sizeof(*A)*3)`. This pass can also introduce additional loads not present in the original program through a form of copy propagation.	no	yes	no	yes	disabled
dse	Performs dead store elimination and redundant store elimination as described in Figure 5.5	no	yes	yes	yes	disabled
x86 Code Generation Passes						
seldag	Builds the initial instruction selection DAG. Performs some CSE during construction.	yes	no	no	no	modified
node-combine	Performs forms of CSE, constant folding, strength reduction, store-to-load forwarding, and dead store elimination on the selection DAG. Can reduce atomicity of certain operations; for instance a store of a 64-bit float that can be done atomically on some architectures may be changed to two 32-bit integer stores. Also, bit-masking code may be recognized and changed to smaller operations without masking. This can have the effect of reordering a store with prior loads.	yes	yes	no	no	modified
scheduling	Schedules machine instructions.	yes	no	no	no	modified
machine-sinking	Sinks load instructions and dependent computation to successor blocks when possible to avoid execution on code paths where they are not used.	yes	no	no	no	modified

loop rotation, loop unswitching, loop unrolling, unreachable code elimination, virtual-to-physical register allocation, and stack slot coloring.

Other LLVM optimizations can relax the order of memory operations. Table 5.1 lists these optimization passes and classifies the kinds of relaxations that are possible in each. To ensure that the compiler would be SC-preserving, a few of these passes were disabled and the remaining passes were modified to avoid reordering accesses to potentially shared memory.

5.3.2 Implementation

The SC-preserving compiler does not perform any heavyweight and/or whole-program analyses to establish whether or not a location is shared (e.g., thread-escape analysis). Simple, conservative, local information is used to decide if a location is potentially shared. During an early phase of compilation, LLVM converts loads and stores of non-escaping function-local variables into reads and writes of virtual registers. Operations on these virtual registers can be freely reordered. In certain situations, structures that are passed by value to a function are accessed using load and store operations. The SC-preserving compiler recognizes these situations and allows these memory operations to be reordered in any sequentially valid manner. In addition, shared memory operations may be reordered with local operations. Thus, for instance, it is safe to allow the "instcombine" pass to transform `t=X; t+=u; t+=X;` into `t=X≪1; t+=u;` when both `t` and `u` are local variables.

Incorporating the necessary modifications into LLVM was a fairly natural and noninvasive change to the compiler code. LLVM already avoids reordering and removing loads and stores marked as being *volatile*. Therefore, in the IR optimization passes existing code written to handle volatiles could often be reused

```
float Distance(float* x,
    float* y, int n)
{
  float sum = 0;
  int i=0;

  for(i=0; i<n; i++){
     sum += (x[i]-y[i])
           *(x[i]-y[i]);
     // Note: x[i] is *(x+i*4)
     //  and  y[i] is *(y+i*4)
  }

  return sqrt(sum);
}
```

(a)

```
float Distance(float* x,
    float* y, int n)
{
  register float sum = 0;
  register px = x;
  register py = y;
  register rn = n;

  for(; rn-->0; px+=4,py+=4){
    sum += (*px-*py)
          *(*px-*py);
  }

  return sqrt(sum);
}
```

(b)

```
float Distance(float* x,
    float* y, int n)
{
  register float sum = 0;
  register px = x;
  register py = y;
  register rn = n;

  for(; rn-->0; px+=4,py+=4){
    register t = (*px-*py);
    sum += t*t;
  }

  return sqrt(sum);
}
```

(c)

Figure 5.6: Example demonstrating optimizations allowed in an SC-preserving compiler. The function in (a) computes the distance between two `n`-dimensional points `x` and `y` represented as arrays. An SC-preserving compiler is able to safely perform a variety of optimizations, leading to the version in (b). However, it cannot eliminate the common-subexpression `*px - *py` involving possibly-shared accesses to the array elements. A traditional optimizing compiler does not have this restriction and is able to generate the version in (c).

in order to restrict optimizations on other accesses to shared memory. The primary mechanism by which reordering was avoided during the x86 code generation passes was by "chaining" memory operations to one another in program order in the instruction selection DAG. This indicates to the scheduler and other passes that there is a dependence from each memory operation to the next and prevents them from being reordered.

5.3.3 Example

The example in Figure 5.6 helps illustrate why an SC-preserving compiler can still optimize programs effectively. The source code shown in part (a) of the figure is a simplified version of a performance-intensive function in one of the bench-

marks. The function calculates the distance between two n-dimensional points represented as (possibly shared) arrays of floating point values. In addition to performing the floating point operations that actually calculate the distance, directly translating this function into x86 assembly would allocate space on the stack for the locally declared variables and perform four address calculations during each iteration of the loop. Each address calculation involves an integer multiply and an integer add operation as hinted by the comments in Figure 5.6 (a). The SC-preserving compiler is able to perform a variety of important optimizations on this code:

- Since the locally declared variables (including the parameters) do not escape the function, they can be stored in registers rather than on the stack.
- CSE can be used to remove two of the address calculations since they are redundant and only involve locals.
- Loop-induction-variable strength reduction allows us to avoid the multiplication involved in the two remaining address calculations by replacing the loop variable representing the array index with a loop variable representing an address offset that starts at zero and is incremented by 4 each iteration.
- Using loop-invariant code motion (and associativity of addition), we can increment the array addresses directly during each iteration rather than incrementing an offset and later adding it to the base addresses.

The final result of applying the above SC-preserving optimizations is shown in part (b) of the figure (using C syntax rather than x86 assembly). The fully optimizing compiler that does not preserve SC is able to perform one additional optimization: it can use CSE to eliminate the redundant floating point loads and

subtraction in each iteration of the loop. The resulting code is shown in part (c) of the figure.

5.3.4 Evaluation

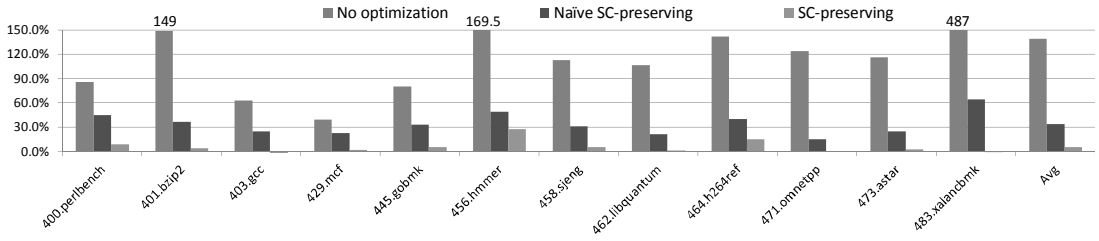

Figure 5.7: Performance overhead incurred by various compiler configurations compared to the stock LLVM compiler with `-O3` optimization for SPEC CINT2006 benchmarks.

This section evaluates the SC-preserving compiler on a variety of sequential and parallel benchmarks. Even though sequential consistency only concerns multi-threaded programs, sequential benchmarks are included in this evaluation since optimizing compilers are tuned to perform well for these benchmarks. The experimental results indicate that the vast majority of the optimizations in LLVM responsible for good performance are in fact SC-preserving.

All programs were executed on an Intel Xeon machine with eight cores, each of which supports two hardware threads and 6 GB of RAM. Each program was evaluated using three compiler configurations. The configuration "No optimization" is the stock LLVM compiler with all optimizations disabled; "Naïve SC-preserving" enables only those LLVM passes that are already SC-preserving, because they never reorder accesses to shared memory; and "SC-preserving" is the full SC-preserving compiler, which includes modified versions of some LLVM passes.

Figure 5.7 shows the results for the SPEC CINT2006 benchmarks. The figure shows the performance overhead of each benchmark under the three compiler

configurations, normalized to the benchmark's performance after being compiled with the stock LLVM compiler and all optimizations enabled (`-O3`). With no optimizations, the benchmarks incur an average 140% slowdown. Re-enabling just the optimizations guaranteed to preserve SC reduces this overhead all the way to 34%. The full SC-preserving compiler reduces the average overhead to only 5.5%, with a maximum overhead for any benchmark of 28%.

The results for parallel applications from the SPLASH-2 and PARSEC benchmark suites are shown in Figure 5.10 from Section 5.6 (the last two compiler configurations shown in the figure pertain to the notion of interference checks that will be introduced in the next section). The results agree with those of the sequential benchmarks. Without optimizations the benchmarks incur an average 153% slowdown. Re-enabling "naïvely SC" optimizations reduces the overhead to 22%, and the full SC-preserving compiler incurs an average overhead of only 2.7%, with a maximum overhead for any benchmark of 34%.

5.4 Speculation for SC-Preservation

As shown in Table 5.1, most of the optimization passes that reorder shared memory accesses only relax the `L` $\rightarrow$ `L` and `S` $\rightarrow$ `L` orderings. In other words, these optimizations have the potential to perform eager loads but no other memory reorderings. In order to evaluate how important these eager-load optimizations are for performance, experiments for the parallel benchmarks were run using the SC-preserving compiler with the four (SC-violating) eager-load IR optimization re-enabled. The "Only-Eager-Loads" configuration in Figure 5.10 illustrates the results. The benchmark with the largest overhead in the SC-preserving compiler, `facesim`, rebounded from a 34% slowdown to only a 6.5% slowdown, and many other benchmarks regained of all of their lost performance.

This experiment motivates the desire to *speculatively* perform eager-load optimizations and then dynamically recover upon a possible SC violation in order to preserve SC. This section describes how the SC-preserving compiler can perform such speculation via a notion of *interference checks*, which conservatively determine whether a memory location's value has been modified since it was last read by the current thread. First the instruction set architecture (ISA) extensions in the hardware that support interference checks will be presented. Then, a strategy for using these new instructions to speculatively perform eager-load optimizations in the compiler is described.

5.4.1 ISA Extensions

Interference checks rely on three new instructions to be provided by the architecture: `m.load` (monitored load), `m.store` (monitored store), and `i.check` (interference check). The `m.load` and `m.store` instructions behave as regular loads and stores but additionally instruct the processor to start monitoring possible writes to the memory location being accessed. We assume that the processor can monitor up to a maximum of N locations simultaneously. These instructions therefore take as an additional parameter a tag from 0 to $N - 1$, which is used as an identifier for the memory location being monitored.

The `i.check` instruction provides a mechanism to query the hardware as to whether or not writes could have occurred to a set of memory locations. It accepts an N-bit mask and a recovery branch target as a parameter. The instruction conditionally branches to the recovery target based on whether or not writes may have occurred for any of the monitored memory addresses indicated by the mask. If the instruction does not branch, it is guaranteed that no thread has written to any of the locations indicated by the mask since the instructions that initiated

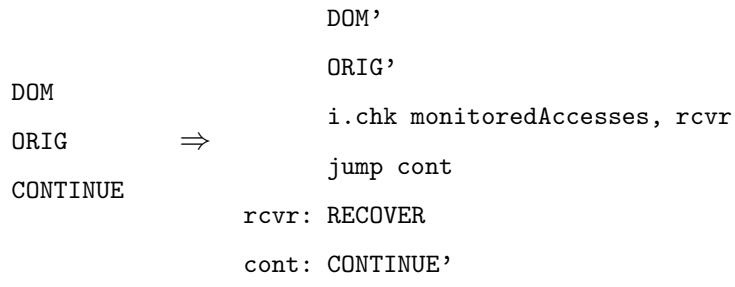

Figure 5.8: Introducing interference checks when performing eager-load transformations in `ORIG`, a single-entry, single-exit region of code with no stores. Either or both of `DOM'` and `ORIG'` contain the definitions for `monitoredAccesses` for the eager loads involved in the transformation.

their monitoring were executed. When using an `i.check` in the examples below, we will list the tags explicitly for clarity rather than using a bit mask.

Note that the use of tags to identify accesses, rather than simply identifying them with the address they access, allows the compiler to safely and accurately use interference checks in the face of potential aliasing. The compiler may end up simultaneously monitoring two accesses to the same location using separate tags due to unknown aliasing. The hardware will correctly report interference between the time when the monitored access for each tag was executed and the time of the `i.check` for that tag. This design places the burden on the compiler to manage the hardware resources available for monitoring. It must ensure that when it reuses a tag, the access that was previously assigned to that tag no longer needs to be monitored.

5.4.2 Interference Check Algorithm

Figure 5.8 illustrates how the compiler performs eager load optimizations with interference checks. In the figure, `ORIG` represents a store-free block of code in which we would like to perform eager load optimizations (e.g., eliminating a load and using a value loaded or stored earlier). `DOM` represents code that

dominates `ORIG` and which may contain accesses we would like to monitor and reuse when performing the optimization. `CONTINUE` is the code that follows `ORIG`. Performing SC-preserving eager load optimizations may require transforming all of these blocks of code, as well as adding a block of code called `RECOVER` which is essentially a copy of the original `ORIG` block to be used when potential interference occurs. Informally, the algorithm works on code in Static Single Assignment form (SSA)[CFR91] in the following steps:

1. Find a contiguous, single-entry, single-exit block of code without stores. Call this block `ORIG`.

2. Create a branch target at the first instruction after `ORIG`. Call the following instructions, starting at this new target, `CONTINUE`.

3. Make a copy of `ORIG` in its entirety and call it `RECOVER`. Note that, since we are manipulating SSA code, all local and temporary values will be given a new SSA name in the copied code.

4. Apply eager-load transformations in `ORIG` and call the resulting block of code `ORIG'`. The transformations may include any combination of the following:

   (a) Eliminate a load and replace its uses with a value from a previous load or store to that address that dominates the current load. This prior memory access may or may not be in `ORIG`. Convert this previous memory access to an `m.load` or `m.store` if it is not already one. If multiple definitions reach the load to be removed, all of them have to be converted.

   (b) Hoist a load from `ORIG` to a position dominating all of its uses, potentially reordering with previous load and/or store operations. Its new

position may or may not be in `ORIG`. Convert the hoisted load to an `m.load`.

We'll call the code that dominated `ORIG` and may now contain monitored instructions `DOM'`. Each access that is converted to a monitored instruction must use a distinct tag, so the compiler is limited to at most N eager-load conversions in this step.

5. Perform any desired SC-preserving optimizations on the code remaining in `ORIG'`.

6. Insert an `i.check` instruction after `ORIG'` that checks for interference on all accesses that were marked as monitored by step 4 and branches to the recovery code on failure.

7. For each value that is live-out of `ORIG`, transform `CONTINUE` by inserting an SSA phi instruction at the beginning that chooses the appropriate value based on whether code flowed from `ORIG'` or `RECOVER`. Call the transformed block `CONTINUE'`.

5.4.3 Implementation and Example

The SC-preserving compiler implementation modifies LLVM's global value numbering (GVN) pass to make use of interference checks in order to allow more aggressive optimization while maintaining SC. The GVN pass performs a variety of eager-load optimizations, including CSE, partial redundancy elimination, and copy/constant propagation. The technique could also apply to other passes that perform eager load optimizations..

Figure 5.9 shows some LLVM IR code that calculates $\texttt{X}^2+\texttt{Y}+\texttt{X}^2$, along with the transformations that take place on it during the GVN pass in order to eliminate

the redundant computation of $\texttt{X}^2$. Virtual registers, or temporaries, are prefixed by the `%` symbol and are in SSA form. First, the GVN pass removes the second load of memory location `X` (which defines `%5`) and replaces all of its uses with the first load of `X`. After this load elimination, we are left with the code in (b), where it is clear that the second `mul` instruction is unnecessary, so it is removed and its use is replaced with the previously calculated value in virtual register `%2`. The final code with the load and multiply eliminated is shown in (c). Figure 5.9(d) shows the result of applying the above algorithm to add interference checks and make this transformation SC-preserving.

5.4.4 Correctness of the Algorithm

Let us now establish that the above algorithm for inserting interference checks is SC-preserving. First consider the case when the interference check fails. Neither `ORIG` nor `ORIG'` contains any stores. Thus, the state of non-local memory does not change during the execution of `ORIG'`. As the code is in SSA form, all the effects of `ORIG'` on local state become dead once the code switches to `RECOVER`, which is a copy of `ORIG`. Hence, other than needlessly executing `ORIG'`, the transformed program has the same behavior as the original program when the interference check fails.

Now consider the case when the interference check succeeds. This means that each monitored memory location is guaranteed to be unmodified from the start of monitoring through the execution of `ORIG'`. The key property of the algorithm is that every memory location involved in an eager load is monitored from the point where the eager load occurs until at least the point at which the load would have occurred in the original program (since it would have occurred somewhere within `ORIG`). Thus the value loaded in the optimized code is the value that would

have been read by the original program, thereby preserving SC.

5.5 Hardware Support for Interference Checks

In this section hardware support for efficiently implementing the `m.load`, `m.store`, and `i.check` instructions is described. The hardware changes required are simple and efficient, and therefore practical. In fact, the newly proposed instructions are similar to the data speculation support in the Itanium's ISA [Ita02], which was designed to enable speculative optimizations in a single thread in the face of possible memory aliasing. The design safely supports both the goal of ensuring sequential consistency as well as Itanium's speculative load optimizations. The required hardware support is simple: a structure to store N addresses (32 in this implementation), each with an associated bit indicating whether the address was possibly written.

5.5.1 Hardware Design

Interference checks are supported using a structure called the Speculative Memory Address Table (SMAT) which is similar to the Advanced Load Address Table (ALAT) used in Itanium processors [Ita02]. SMAT is a Content-Addressable-Memory (CAM). It has N entries, enabling the compiler to monitor interference on N addresses at any instant of time. Each entry in the SMAT contains an address field and an interference bit.

We collectively refer to `m.load` and `m.store` instructions as *monitor* instructions. As described in the previous section, each monitor instruction contains a tag between 0 and $N-1$. When executing a monitor instruction, the hardware stores the address accessed by that instruction in the SMAT entry specified by the

tag, resets that entry's interference bit, starts to monitor writes to the address, and executes the memory operation requested by the instruction.

A processor core can easily detect when another processor core wants to write an address by monitoring invalidation coherence requests. When a processor core receives an invalidation to a cache block, the interference bit of each SMAT entry holding an address from that block is set. The interference bit of an entry is also set when a store to the associated address commits from the current processor. While the latter behavior is not necessary to preserve SC, it enables Itanium-style speculative load optimizations [Ita02].

The compiler generates an `i.check` instruction with an N-bit `mask` to check for interference on a maximum of N different addresses. Each bit in the `mask` corresponds to an entry in the SMAT. The hardware executes the `i.check` instruction by checking the interference bits of the SMAT entries specified in its `mask`. If any of the checked interference bits is set, the hardware branches to the recovery code whose target is specified in the `i.check` instruction.

The hardware updates the SMAT for a monitor instruction and executes `i.check` instructions only when they are ready to commit from a processor core's instruction window. This ensures that the hardware does not update SMAT entries speculatively while executing instructions on an incorrect path taken due to branch misprediction. Nevertheless, the semantics of these instructions requires that the hardware notice interference from the moment the instruction executes (i.e., when an `m.load` receives its value from the cache). An existing hardware mechanism, described in §5.5.2, can be relied upon to perform the monitoring between the time an `m.load` executes and the time it commits from the ROB.

5.5.2 Relation To In-Window Hardware Speculation

The approach of monitoring invalidation coherence requests to detect interference for a set of addresses is similar to what many processors already implement for efficiently supporting TSO at the hardware level [GGH91]. TSO does not allow a load to be executed before another load in program order even if they are accessing different addresses. To achieve good performance, Gharachorloo et al. [GGH91] proposed to speculatively execute loads out-of-order. However, instructions are still committed in order from a FIFO queue called the reorder buffer (ROB). Therefore, to detect misspeculation the hardware simply needs to detect when another processor tries to write to an address that has been read by a load that is yet to commit from the ROB. This is achieved by monitoring the address of invalidation coherence requests from other processor cores. On detecting a misspeculation, the hardware flushes the misspeculated load and its following instructions from the pipeline and restarts execution.

The proposed hardware design essentially extends the above hardware mechanism to detect interference for addresses of certain memory operations (specified by the compiler) even after they are committed from the ROB. This allows a compiler to eagerly execute loads and later check for interference at the original location of the load in the source code. On an `m.load`, the monitoring needs to start logically when the processor receives the value of the load. However, the SMAT entry is updated only when the instruction is committed. In between the two events, when the load instruction is in flight in the ROB, the monitoring performed above is sufficient to provide the required semantics of the `i.check`.

5.5.3 Conservative Interference Checks

While an implementation of interference checks must detect interference whenever it occurs, it is legal to signal interference when none actually exists. Such false positives are acceptable in this design because they simply result in execution of the unoptimized code, losing some performance but maintaining SC. The ability to tolerate false positives avoids a number of potentially complex issues and keeps the hardware simple.

First, the hardware monitors interference at the cache block granularity as coherence invalidation messages operate at cache block level. This may result in false positives when compared to a detector that monitors byte-level access. But the probability that a cache block gets invalidated between a monitor instruction and an `i.check` is very low. Moreover, frequent invalidations or "false sharing" of hot cache lines result in performance degradations and thus can expected to be rare in well-tuned applications.

Second, SMAT entries for a cache block that gets evicted due to capacity constraints are conservatively invalidated. Monitoring interference for uncached blocks would require significant system support (similar in complexity to unbounded transactional memory systems [CNV06]).

Third, in ISAs like `x86` one memory instruction could potentially access two or more cache lines, but a SMAT entry can monitor only one cache block address. To address this problem, if a monitor instruction accesses more than one cache block, the interference bit for the SMAT entry is immediately set. This could cause a future `i.check` to fail forcing execution down an unoptimized path. Fortunately, such unaligned cache accesses are rare.

Finally, a context switch may occur while multiple addresses are monitored

in the hardware SMAT. Instead of virtualizing this structure, the interference bit in all SMAT entries is set after a context switch. This may cause future `i.check` instructions from a thread to fail unnecessarily when it is context switched back in, but this overhead is likely to be negligible since context switches are relatively rare when compared to the frequency of memory accesses.

5.6 Results

The experimental results relating to the performance of the base SC-preserving compiler were discussed in Section 5.3.4. In this section we discuss additional experiments which evaluate the potential effectiveness of using interference checks. In addition, the performance of the SC-preserving compilers is compared to a fully optimizing compiler running on simulated hardware that uses a DRF0 memory model which is more relaxed (allows more hardware reorderings) than TSO. This gives a sense of the performance burden of providing a strong, end-to-end memory model across hardware and software.

5.6.1 Compiler Configurations

As described in Section 5.3.4, the baseline compiler is the out-of-the-box LLVM compiler with all optimizations (`-O3`). The experiments on parallel benchmarks use the three compiler configurations discussed in that section ("No optimization", "Naïve SC-preserving", and "SC-preserving"), as well as two additional configurations. The "Only Eager Loads" configuration includes all the optimizations from the SC-preserving compiler plus the unmodified (SC-violating) version of all IR passes that perform only eager load optimizations (GVN, instcombine, arg-promotion, and jump-threading). This configuration is intended to give a sense

Table 5.2: Baseline IPC for simulated DRF0 hardware running binaries from the stock LLVM compiler.

Application	Avg. IPC	Application	Avg. IPC
blackscholes	1.94	bodytrack	1.61
fluidanimate	1.28	swaptions	1.67
streamcluster	1.42	barnes	1.57
water(nsquared)	1.66	water(spatial)	1.66
cholesky	1.78	fft	1.39
lu(contiguous blocks)	1.64	radix	0.99

of the opportunity for improvement available to optimizations based on the interference check technique and is only used for experiments on native hardware and not on simulated machines. Finally, the "SC-preserving+GVN w/ ICheck" configuration includes all of the optimizations from the SC-preserving compiler plus a modified GVN pass that is made SC-preserving using interference checks and recovery code. When this configuration targets a simulated machine with appropriate support, it emits `m.load`, `m.store`, and `i.check` instructions. But when it targets native hardware, the configuration emits `m.load` and `m.store` instructions as regular loads and stores and emulates a never-failing `i.check` using a logical comparison of constant values followed by a conditional branch. Thus, when running on the native machine, the overhead caused by increased code size and the additional branch is captured, but the effect of actual or false conflicts on monitored accesses is not. In a real implementation, however, we expect the `i.check` instruction to be more efficient than a branch.

5.6.2 Benchmarks

The performance of the various compiler configurations was evaluated on the PARSEC [BKS08] and SPLASH-2 [WOT95] parallel benchmark suites. Table 5.2 lists the average instructions executed per cycle (IPC) for each of these bench-

marks when compiled with the stock LLVM compiler at `-O3` optimization and run on simulated DRF0 hardware which implements weak consistency and is described below. All of these benchmarks are run to completion. For experiments on actual hardware, the `native` input for PARSEC benchmarks was used, while for the simulated machines, the `sim-medium` input set was used to keep the simulation time reasonable. (Since `streamcluster` was especially slow to simulate, the `sim-small` input was used.) For SPLASH-2 applications, the default inputs were used for simulation while inputs were modified to increase the problem size for experiments on native hardware. Correct behavior of the benchmarks under all compiler configurations was verified by using a self-testing option when available, or by comparing results with those produced when compiling the benchmark using gcc.

5.6.3 Experiments on Native Hardware

All six compiler configurations (including the baseline) were evaluated on an Intel Xeon machine with eight cores, each of which supports two hardware threads and 6 GB of RAM. Each benchmark was run five times for each compiler configuration and the execution time was measured. (The results given here are for CPU user time, though the results for total time elapsed were very similar.) The overheads given are relative to the baseline, fully-optimizing compiler and are shown in Figure 5.10. Let's consider the base SC-preserving compiler first. Notice that for many of the benchmarks, restricting the compiler to perform only SC-preserving optimizations has little or no effect. In fact, in some cases, disabling these transformations appears to speed the code up, indicating that the compiler ought not to have performed them in the first place. There are several benchmarks, however, for which the SC-preserving compiler incurs a

noticeable performance penalty, 34% in the case of `facesim`.[1] On average, we see a 2.7% slowdown. Consider now the compiler configuration which re-enables various eager load optimizations. Several of the applications which suffered a significant slowdown under the SC-preserving compiler regain much of this performance in this configuration. Most notably, `facesim` vastly improves to 6.5% and `bodytrack`, `streamcluster`, and `x264` recover all (or nearly all) of their lost performance. On average, the compiler with eager load relaxations enabled is as fast as the stock compiler, indicating that the technique of using interference checks to safely allow eager load optimizations holds significant promise. Finally, the rightmost bar in the graph shows the slowdown of the aggressive SC-preserving compiler that includes the modified GVN pass with interference checks. (Remember, we are running on a native machine in this set of experiments, so a never-fail load check is emulated.) This technique regains a good portion of the performance lost by the base SC-preserving compiler for `facesim`, reducing the overhead from 34% to under 20%, with `streamcluster` and `x264` showing a more modest improvement.

5.6.4 Experiments on Simulated Machines

To study the performance of interference checks in hardware, the benchmarks were run on a cycle-accurate, execution driven, Simics [MCE02] based x86_64

[1] Additional profiling and investigation revealed that the slowdown in `facesim` was largely caused by a commonly invoked 3x3 matrix multiply routine. The SC-preserving compiler was unable to eliminate the two redundant loads of each of the 18 shared, floating point matrix entries involved in the calculation. This resulted in 36 additional load instructions for each matrix multiplication performed by the SC-preserving version of `facesim`. The modified GVN pass with interference checks is able to relegate the 36 additional loads to the recovery code, eliminating them on the fast path. A straightforward rewrite of the source code to first read the 18 shared values into local variables would have allowed the base SC-preserving compiler to generate the fully optimized code.

Table 5.3: Simulated processor configuration for evaluation of SC-preserving compiler with interference checks.

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 directory protocol
Interconnection	Hierarchical switch, 10 cycle hop latency.
Memory	80 cycle DRAM lookup latency.
SMAT	32 entries CAM structure, 1 cycle associative lookup

simulator called FeS2 [FeS]. The benchmarks were also run on simulated TSO hardware, with and without support for interference checks, and compared it to DRF0 hardware that supports weak consistency. The processor configuration that was modelled is shown in Table 5.3. For the TSO simulation, a FIFO store buffer that holds pending stores and retires them in-order was used. Speculative load execution support [GGH91] was also modelled. The weakly consistent DRF0 simulation allowed stores and loads to retire out-of-order.

Figure 5.11 shows the results of the simulation study. When compared to the fully optimizing compiler configuration running on the simulated DRF0 machine, the performance overhead of using the SC-preserving compiler on simulated TSO hardware is 3.4% on average. This cost is reduced to 2.2% when the GVN pass with interference checks is used. For several programs that incur significant overhead, such as `bodytrack` and `facesim`, the interference check optimizations reduce the overhead to almost zero. For `streamcluster`, the overhead is reduced from about 23% to 17%. The frequency of load-check failures is, on average, only about one in ten million instructions. This indicates that the performance overhead due to false positives arising from several hardware simplifications described

in Section 5.5.3 is negligible.

5.7 Conclusion

Sequential consistency is an intuitive memory model, but is widely believed to be difficult or impossible to implement efficiently. The research in this chapter empirically demonstrates that the performance incentive for relaxing SC semantics in the compiler is much less than previously assumed. Building an SC-preserving compiler required only simple modifications to LLVM, a state-of-the-art C/C++ compiler. For a wide range of programs from the SPLASH-2, PARSEC, and SPEC CINT2006 benchmark suites, the SC-preserving compiler results in a performance overhead of only 3.8% on average with a maximum of 34% overhead.

In the case where even small overheads are unacceptable, targeted, fine-grained, compiler-directed, conservative race detection can enable additional optimizations while still preserving SC semantics. Interference checks are an implementation of such a detection mechanism. They allow the SC-preserving compiler to regain much of the performance lost due to restrictions on SC-violating compiler optimizations.

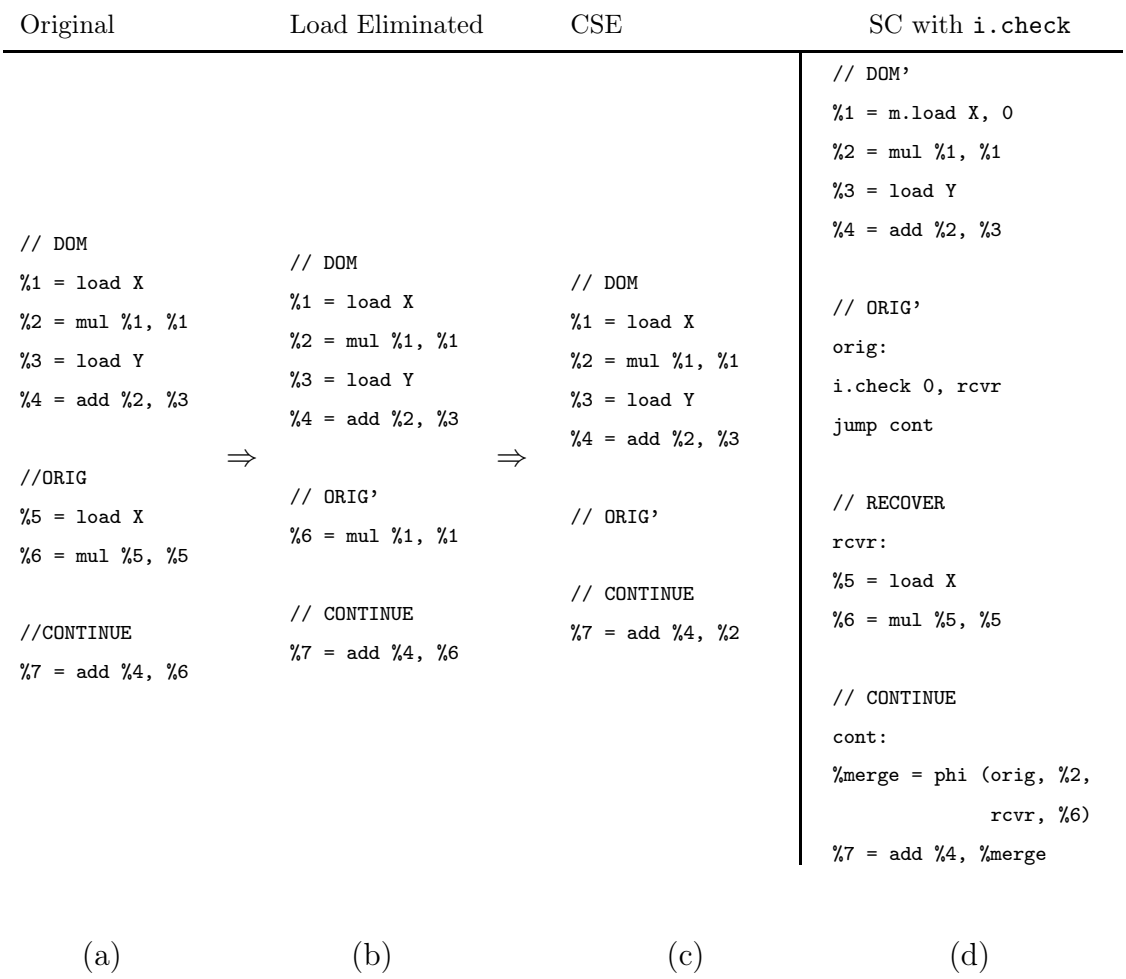

Figure 5.9: GVN first transforms program (a) into (b) by eliminating the "available load" from `X`, then notices that the result of the second multiplication has already been computed and performs common subexpression elimination to arrive at (c). This transformation is not SC since it reorders the second load of `X` with the load of `Y`.

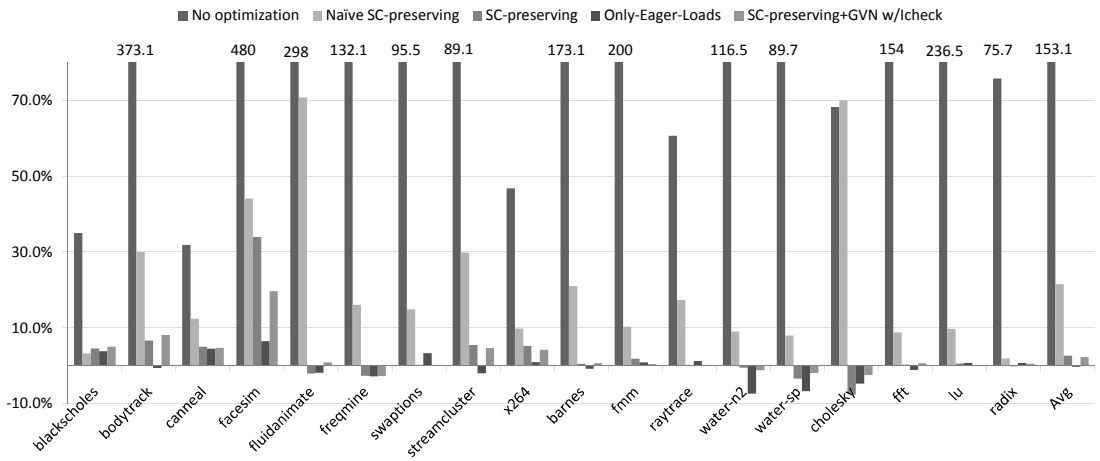

Figure 5.10: Performance overhead incurred by the various compiler configurations compared to stock LLVM compiler with `-O3` optimization running on native Xeon hardware for PARSEC and SPLASH-2 benchmarks.

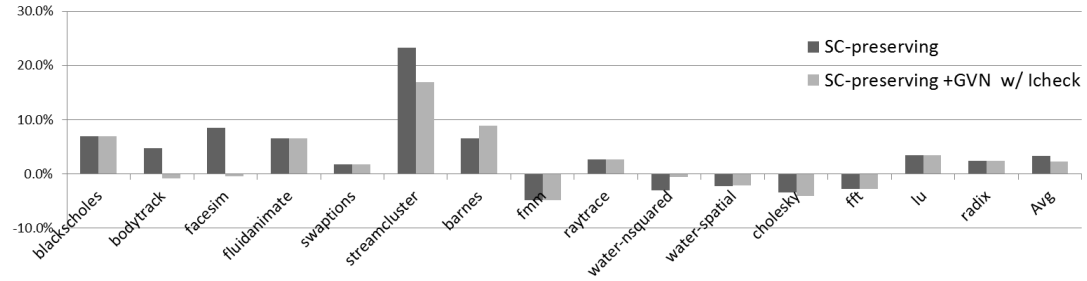

Figure 5.11: Performance overhead of SC-preserving compiler on simulated TSO hardware with and without using interference checks relative to fully optimizing SC-violating compiler on simulated DRF0 hardware.

CHAPTER 6

Related Work

6.1 Data Race Detection

Prior data race detection can be broadly classified into static and dynamic techniques. Static techniques include those that use type-based analysis [BLR02, FF00, SAW05] or data-flow analysis [Ste93, EA03, NAW06, VJL07, PFH06] to ensure that all data accesses are consistently protected by locks. Many of these techniques are scalable and most are complete in that they find *all* data races in a program. The downside is that static techniques are inherently imprecise and typically report a large number of false data races that place a tremendous burden on the user of the tool. More importantly, these techniques are not able to handle synchronizations other than locks, such as events, semaphores, and condition variables common in many systems programs. Thus, data accesses that are synchronized through these mechanisms will be falsely reported as potential data races. Model checking techniques [HJM04, QW04] are capable of handling such synchronizations, but are not scalable due to the complexity of their analysis. Dynamic analyses, like those presented in this dissertation, do not suffer from these problems.

One of the main limitations of dynamic data race detection tools is their high run-time overhead. There have been attempts to ameliorate the performance cost of dynamic analysis using static optimizations for programs written

in strongly typed languages [CLL02]. Dynamic data race detectors for managed code [YRC05] also have the advantage that the runtime system already incurs the cost of maintaining metadata for the objects, which they make use of. For unmanaged code like C and C++, however, the runtime performance overhead of data race detection remains high. Intel's ThreadChecker [SBM06], for example, incurs about 200× overhead to find data races. The detection techniques described in this dissertation target particular problems. As such they can fail to find certain races while still proving useful.

6.1.1 Happened-before versus Lockset Dynamic Detection

Dynamic race detection algorithms can be broadly classified into happened-before based algorithms [Lam78, Net93, AHM91, CMN91, DS90, Cru91, Sch89, PK96, RB00, MC91], lockset based algorithms [SBN97, PG01, Nis04, ASW05] and hybrid algorithms that combine the two [DS91, YRC05, OC03, PS03].

One class of data race detectors use the lockset algorithm. The lockset algorithm checks whether each shared variable in a program is consistently guarded by at least one lock. Eraser [SBN97] implements the lockset algorithm using instrumentation to dynamically find the data races during a program's execution. This algorithm has been extended to object-oriented languages [PS03] and improved for precision and performance [ASW05, Nis04, PG01, CLL02]. The lockset algorithm has the potential to report false positives due to conflicting accesses that are ordered using synchronization mechanisms other than locks. A recent work [LTQ06] reports that a lockset algorithm resulted in thousands of false positives for scientific applications.

Happened-before-based detectors, on the other hand, check whether conflicting accesses to shared variables are ordered by explicit synchronization oper-

ations or not. Many dynamic race detectors implement the happened-before algorithm in software[RB00]. Hardware [MC91, PT03] and Distributed-Shared-Memory [PK96, RL98] implementations have also been proposed to reduce the runtime overhead of these detectors. The advantage of using a happened-before algorithm is that it can detect the data races with no false positives, although for programs that use mainly locks, it may detect a potential data race in the program on fewer executions than the lockset approach would.[1]

LiteRace uses happened-before-based detection rather than lockset because it aims to support applications that use a variety of synchronization primitives, not just locks. Though the sampling technique used in LiteRace could equally well be applied to a lockset-based detector. The detection mechanism used in DRFx can also be thought of as happened-before-based, since any races from concurrently executing regions arise from conflicting accesses that are not ordered by the happened-before relation.

It is also possible to combine the happened-before and lockset algorithms [DS91, YRC05, OC03, PS03] to get coverage and performance close to a lockset algorithm, and at the same time reduce false positives using happened-before relations.

6.1.2 Sampling Techniques for Dynamic Analysis

The technique used in LiteRace was inspired by prior work in dynamic analyses other than data race detection. Arnold et al. [AR01] proposed sampling techniques to reduce the overhead of instrumentation code in collecting profiles for

[1]The lockset algorithm reports a race as soon as the set of potential locks protecting a location is determined to be empty. This can happen even on an execution/interleaving where there is a happened-before relationship between two accesses.

feedback directed optimizations. Chilimbi and Hauswirth proposed an adaptive sampler for finding memory leaks [HC04]. LiteRace extends their solution to the sampling of multi-threaded programs, and shows that samplers can be effectively used to find data races as well. QVM [AVY08] is an extension to the Java Virtual Machine that provides an interface to enable dynamic checking such as heap properties, local assertions, and typestate properties. It uses sampling to trade-off accuracy with runtime overhead. The sampling technique used in QVM is object-centric, in that, all the events to a sampled object's instance are profiled. In contrast, the samplers in LiteRace are based on the cold-region hypothesis.

6.2 Memory Models

6.2.1 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 [SS88]. Krishnamurthy and Yelick [KY96] proved that computing a minimal delay set (i.e., set of fences) for a program is NP-complete. Two recent projects, Titanium [KSY05] and Pensieve [SFW05], extend the delay set algorithm to reduce the number of fences needed to guarantee SC. In addition, [LNG10] describes a new hardware mechanism called a *conditional fence* that can reduce the cost of executing all of the inserted fences in hardware. These analyses leverage a number of techniques to determine whether a memory location can potentially be involved in a race, including sharing inference [LAY03], 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++. But when applicable, they guarantee end-to-end SC for all programs since the inserted fences prevent reorderings by both the compiler and the hardware.

In contrast, the DRFx model does not require expensive analysis, but does require specialized hardware support. It 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. But, it only guarantees SC for data-race-free programs and may terminate racy programs. The SC-preserving compiler avoids introducing SC violations during compilation using simple modular techniques and still manages to optimize effectively, but it does not prevent hardware reorderings from violating SC.

At the hardware level, various forms of speculation have been proposed to reduce the performance overhead of SC [RPA97, BMW09, CTM07]. 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 [AQL09] addresses this problem in the context of Java programs that use locks. The bulk compiler partitions a program into "chunks" and the BulkSC hardware employs speculation and recovery to ensure serializable execution of chunks. Even then, all these hardware proposals above require speculative execution, checkpointing, and rollback in case of conflicts, which tremendously increases the hardware complexity. Unlike DRFx, these proposals require possibly unbounded resources and thus have to include appropriate mechanisms to handle overflow cases.

6.2.2 Always-on Race Detection and Memory Model Exceptions

Prior research has suggested using data race detection as a way to terminate buggy programs at runtime. Elmas et al. [EQT07] augment the Java virtual machine to dynamically detect bytecode-level data races and raise a `DataRaceException`. Recently, Boehm [Boe09] 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, precisely detecting data races either incurs 8× or more performance overhead in software [FF09] or incurs significant hardware complexity [PT03, MSQ09] despite many proposed optimizations to the basic technique. The large overhead comes from the need to dynamically build the happened-before relation [Lam78] 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).

The DRFx memory model builds on the work of Gharachorloo and Gibbons [GG91], who recognized that it suffices to detect SC violations directly rather than data races. They describe a simple conflict detection algorithm that ensures DRFx's 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 since it ignores the effects of possible compiler reorderings [CDL09, GG91]. DRFx extends their approach with a notion of regions to safely allow such compiler reorderings while still detecting all SC violations.

In concurrent work to our work on DRFx, Lucia et al. [LCS10] 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. The DRFx implementation achieves efficiency in spite of smaller bounded regions by distinguishing soft fences from hard fences and allowing the hardware to optimize across soft fences.

The interference checks in the SC-preserving compiler are a form of dynamic data-race detection that is sufficient to ensure that certain compiler transformations don't violate SC. While the approaches above detect all races that could violate end-to-end SC, interference checks only target SC violations that result from compiler reorderings. However, they have the advantage of being fine-grained, requiring data race detection only for variables that are involved in a compiler optimization and only during the dynamic lifetime of that optimization's effect. Also, the detection is made possible with relatively minimal hardware support based on speculation mechanisms that exist in real hardware [Yea02], rather than requiring the complexity of TM-style conflict detection. Finally, this work shows how to safely recover from interference for common compiler optimizations based on eager loads. This allows the execution to continue while maintaining SC, avoiding the need to throw an exception.

6.2.3 Transactional Memory Systems

Hammond et al. [HWC04] proposed a memory consistency model based on a transactional programming model [HM93]. 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.

The DRFx hardware conflict detection algorithm is similar to the one proposed by Hammond et al. [HWC04] 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.

6.3 Compiler Optimizations

6.3.1 Strengthening Memory Models by Restricting the Compiler

In recent work, Ševčík et al. describe a concurrency extension to a small C-like programming language that provides end-to-end TSO semantics [SVZ11]. They modify an existing compiler for the language and mechanically prove that the optimizations are TSO-preserving, thereby providing an end-to-end guarantee when the resulting binaries are executed on x86 hardware. Our performance measure-

ments complement their work by indicating that a TSO-preserving compiler could be practical to use in a full-fledged programming language.

6.3.2 Optimistic Optimization via Hardware Speculation

The SC-preserving compiler's interference checks are inspired by a common hardware mechanism for enabling out-of-order execution in the presence of strong memory models [GGH91]. This mechanism [Yea02] allows a memory load to be executed out-of-order speculatively, before earlier instructions have completed. Once those instructions have completed, the load need not be re-executed if the value has not changed in the meanwhile, and this can be conservatively detected by checking if the associated cache line has been invalidated. This work demonstrates how the technique can be adapted to the compiler by viewing common compiler optimizations as performing eager (i.e., speculative) reads; a simple interface through which the hardware can expose this mechanism to the compiler is described.

Others have proposed hardware support for dynamically detecting memory aliasing between local loads and stores in a single thread and expose that feature to the compiler so that it can perform optimistic optimizations [GCM94, PGM00]. The Itanium processor implemented this feature using an Advanced Load Address Table (ALAT) to enable aggressive load optimizations [Ita02]. Recently, Nagarajan and Gupta [NG09] extended Itanium's ALAT mechanism to detect memory aliasing with remote writes, enabling the compiler to speculatively reorder memory operations across memory barriers. While interference checks use a mechanism similar to these proposals, they solve a different problem: preserving SC in the face of common compiler transformations.

CHAPTER 7

Conclusion

Data races are a common flaw in shared memory, concurrent programs. They often lead to insidious bugs that are difficult to isolate and fix. Even assuming that programs exhibit sequentially consistent behavior, data races can lead to confusing outcomes. Moreover, modern architectures and programming languages provide relaxed memory models, weaker than sequential consistency, that further complicate reasoning about racy programs.

Detecting data races has been the subject of much research effort, and both static and dynamic techniques exist for finding them. But static approaches report many false positives and have limited applicability while precise dynamic data race detection drastically slows programs making it impractical to apply in many situations. The research in this dissertation aims to make data race detection practical and helpful to programmers of shared memory, concurrent programs. It demonstrates how efficient forms of imprecise dynamic data race detection can be applied to find bugs and to simplify memory models.

LiteRace uses intelligent sampling to make data race detection for the purpose of bug finding efficient. While it can fail to find some races exhibited during an execution, it never reports a false positive, thus easing the burden on programmers and testers. The sampler, based on the cold path hypothesis, manages to find nearly 70% of the races exhibited in a program while analyzing only 2% of the dynamic memory accesses. The average overhead of only 28% makes running

LiteRace on many test executions feasible, thus allowing more data races to be uncovered.

The DRFx memory model uses a novel form of cooperation between the compiler and the hardware in order to provide strong, end-to-end guarantees to programmers. While current DRF0 memory models provide weak or no semantics to racy programs, DRFx provides simple guarantees for executions of all programs: if an execution terminates in a memory model exception, the program has a data race; if an execution terminates normally, it exhibits SC behavior. DRFx efficiently provides this guarantee by targeting data race detection only at concurrently executing, bounded regions of code. By avoiding optimizations across region boundaries in the compiler and the hardware, SC behavior is guaranteed if a race is not detected among regions that execute concurrently, even if other data races in the program go undetected. The detection employed by DRFx hardware must never report a false race, otherwise data-race-free programs could be terminated, violating the guarantee. Experimental results show that a set of benchmarks built using a DRFx-compliant compiler and run on simulated DRFx-compliant hardware incur an average slowdown of only 11% on average.

The SC-preserving compiler calls into question the importance of violating sequential consistency for the sake of performance when optimizing programs in a compiler. Modifying LLVM, a state-of-the-art C and C++ compiler, to be SC-preserving results in only 3.8% overhead on average for a set of benchmarks run on a Xeon multicore processor. Even when certain SC-violating optimizations are needed for performance, in particular eager load optimizations, they can be enabled in an SC-preserving manner by using fine-grained, compiler-directed, dynamic race detection. Recovery code inserted by the compiler allows the program to recover SC behavior when a race is detected dynamically. As such, the detec-

tion used in the SC-preserving compiler can conservatively report a race, unlike the detection schemes for DRFx and LiteRace. Furthermore, unlike in the other schemes, the detection is limited to particular memory locations indicated by interference check instructions inserted by the compiler. The hardware mechanism necessary to implement the targeted race detection needed by the interference checks is straightforward and similar to existing features in real processors.

To summarize, dynamic data race detection can be used to improve the state of the art in shared memory concurrent programming without compromising performance and with reasonable complexity. The key is to relax the requirement that the analysis precisely identify all data races in the execution being monitored. Depending on the way in which this requirement is relaxed, we can achieve race detection that addresses different challenges in the understanding of shared memory systems. Furthermore, the relaxed techniques avoid the hefty performance or complexity penalty normally associated with dynamic data race detection. I presented three techniques supporting this view and hope the research described in this dissertation eventually helps programmers to more easily debug and understand their shared memory, concurrent code.

References

[AG96] S. Adve and K. Gharachorloo. "Shared Memory Consistency Models: a tutorial." *Computer*, **29**(12):66–76, 1996.

[AH90] S. V. Adve and M. D. Hill. "Weak ordering—a new definition." In *Proceedings of ISCA*, pp. 2–14. ACM, 1990.

[AHM91] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. "Detecting data races on weak memory systems." In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer architecture*, pp. 234–243, 1991.

[AQL09] Wonsun Ahn, Shanxiang Qi, Jae-Woo Lee, Marios Nicolaides, Xing Fang, Josep Torrellas, David Wong, and Samuel Midkiff. "BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support." 2009.

[AR01] Matthew Arnold and Barbara G. Ryder. "A framework for reducing the cost of instrumented code." In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2001.

[ASW05] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. "Optimized run-time race detection and atomicity checking using partial discovered types." In *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 233–242, 2005.

[AVY08] M. Arnold, M. Vechev, and E. Yahav. "QVM: An efficient runtime for detecting defects in deployed systems." In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 2008.

[BA08] H. J. Boehm and S. Adve. "Foundations of the C++ concurrency memory model." In *Proceedings of PLDI*, pp. 68–78. ACM, 2008.

[BKS08] C. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications." In *Proceedings of PACT*, October 2008.

[BLR02] C. Boyapati, R. Lee, and M. Rinard. "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks." In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 211–230, 2002.

[BMW09] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. "InvisiFence: Performance-transparent memory ordering in conventional multiprocessors." In *Proceedings of the 36th annual International Symposium on Computer architecture*, ISCA '09, pp. 233–244. ACM, 2009.

[Boe09] H. J. Boehm. "Simple thread semantics require race detection." In *FIT session at PLDI*, 2009.

[BOS11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. "Mathematizing C++ concurrency." In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pp. 55–66. ACM, 2011.

[Bus] Lawrence Bush. "Generic Concurrent Lock-free Linked List." `http://people.csail.mit.edu/bushl2/rpi/project_web/page5.html`.

[CB01] M. Christiaens and K. De Bosschere. "TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs." In *Proceedings of the Java Virtual Machine Rsearch and Technology Symposium (JVM'01)*, 2001.

[CDL09] L. Ceze, J. Devietti, B. Lucia, and S. Qadeer. "The Case for System Support for Concurrency Exceptions." In *USENIX HotPar*, 2009.

[CFR91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." *ACM Transactions on Programming Languages and Systems*, **13**(4):451–490, Oct 1991.

[CKS07] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. "The Java Memory Model: Operationally, Denotationally, Axiomatically." In *ESOP*, pp. 331–346, 2007.

[CLL02] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. "Efficient and precise datarace detection for multithreaded object-oriented programs." In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 258–269, 2002.

[CMN91] J. D. Choi, B. P. Miller, and R. H. B. Netzer. "Techniques for debugging parallel programs with flowback analysis." *ACM Transactions on Programming Languages and Systems*, **13**(4):491–530, 1991.

[CNV06] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. "Unbounded page-based transactional memory." *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[Cru91] J. M. Crummey. "On-the-fly detection of data races for programs with nested fork-join parallelism." In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 24–33, 1991.

[CTM07] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. "BulkSC: bulk enforcement of sequential consistency." In *ISCA*, pp. 278–289, 2007.

[CTT06] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. "Bulk disambiguation of speculative threads in multiprocessors." In *ISCA '06*, 2006.

[DLM09] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. "Early experience with a commercial hardware transactional memory implementation." In *Proceedings of ASPLOS*, 2009.

[DS90] A. Dinning and E. Schonberg. "An empirical comparison of monitoring algorithms for access anomaly detection." In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pp. 1–10, 1990.

[DS91] A. Dinning and E. Schonberg. "Detecting access anomalies in programs with critical sections." In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pp. 85–96, 1991.

[Duf07] Joe Duffy. "A query language for data parallel programming: invited talk." In *DAMP*, p. 50, 2007.

[EA03] D. Engler and K. Ashcraft. "RacerX: Effective, static detection of race conditions and deadlocks." In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 237–252, 2003.

[EQT07] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. "Goldilocks: a race and transaction-aware java runtime." In *Proceedings of PLDI*, pp. 149–158, 2007.

[FeS] "The FeS2 simulator." `http://fes2.cs.uiuc.edu`.

[FF00] C. Flanagan and S. N. Freund. "Type-based race detection for Java." In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 219–232, 2000.

[FF09] C. Flanagan and S.N. Freund. "FastTrack: efficient and precise dynamic race detection." In *Proceedings of PLDI*, 2009.

[GCM94] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. "Dynamic Memory Disambiguation Using the Memory Conflict Buffer." In *ASPLOS*, pp. 183–193, 1994.

[GF02] Chris Gniady and Babak Falsafi. "Speculative Sequential Consistency with Little Custom Storage." In *IEEE PACT*, pp. 179–188, 2002.

[GG91] K. Gharachorloo and P.B. Gibbons. "Detecting violations of sequential consistency." In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pp. 316–326. ACM New York, NY, USA, 1991.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. "Two techniques to enhance the performance of memory consistency models." In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 1, pp. 355–364, 1991.

[GLL90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory consistency and event ordering in scalable shared-memorymultiprocessors." In *Proceedings of ISCA*, pp. 15–26, 1990.

[HC04] M. Hauswirth and T. M. Chilimbi. "Low-overhead memory leak detection using adaptive statistical profiling." In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pp. 156–164, New York, NY, USA, 2004. ACM.

[HCW04] 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: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pp. 1–13, New York, NY, USA, 2004. ACM Press.

[Hen06] John L. Henning. "SPEC CPU2006 benchmark descriptions." *SIGARCH Computer Architecture News*, **34**:1–17, September 2006.

[Hil98] Mark D. Hill. "Multiprocessors Should Support Simple Memory-Consistency Models." *IEEE Computer*, **31**:28–34, 1998.

[HJM04] T. A. Henzinger, R. Jhala, and R. Majumdar. "Race checking by context inference." In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pp. 1–13, 2004.

[HM93] Maurice Herlihy and J. Eliot B. Moss. "Transactional memory: architectural support for lock-free data structures." In *Proceedings of ISCA*, pp. 289–300. ACM, 1993.

[HWC04] Lance Hammond, Vicky Wong, Michael K. Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. "Transactional Memory Coherence and Consistency." In *ISCA*, pp. 102–113, 2004.

[IBY07] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: Distributed data-parallel programs from sequential building blocks." In *Proceedings of the EuroSys Conference*, pp. 59–72, 2007.

[Ita02] "Inside the Intel Itanium 2 Processor." *Hewlett Packard Technical White Paper*, 2002.

[KSY05] A. Kamil, J. Su, and K. Yelick. "Making sequential consistency practical in Titanium." In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 15. IEEE Computer Society, 2005.

[KY96] A. Krishnamurthy and K. Yelick. "Analyses and optimizations for shared address space programs." *Journal of Parallel and Distributed Computing*, **38**(2):130–144, 1996.

[LA04] 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.

[Lam78] L. Lamport. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM*, **21**(7):558–565, 1978.

[Lam79] L. Lamport. "How to make a multiprocessor computer that correctly executes multiprocess programs." *IEEE transactions on computers*, **100**(28):690–691, 1979.

[LAY03] B. Liblit, A. Aiken, and K. Yelick. "Type systems for distributed data sharing." In *Proceedings of the Tenth International Static Analysis Symposium*, 2003.

[LCS10] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans Boehm. "Conflict Exceptions: Providing Simple Parallel Language Semantics with Precise Hardware Exceptions." June 2010.

[Lev93] Nancy G. Leveson. "An Investigation of the Therac-25 Accidents." *IEEE Computer*, **26**:18–41, 1993.

[LNG10] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. "Efficient sequential consistency using conditional fences." In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pp. 295–306. ACM, 2010.

[LTQ06] S. Lu, J. Tucek, F. Qin, and Y. Zhou. "AVIO: detecting atomicity violations via access interleaving invariants." In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 37–48, 2006.

[MC91] S. L. Min and J.-D. Choi. "An Efficient Cache-Based Access Anomaly Detection Scheme." In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pp. 235–244, 1991.

[MCE02] S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. "Simics: A Full System Simulation Platform." *IEEE Computer*, **35**(2):50–58, 2002.

[Mica] Microsoft. "Phoenix Compiler." *http://research.microsoft.com/Phoenix/*.

[Micb] Microsoft. "Thread Execution Blocks." *http://msdn.microsoft.com/en-us/library/ms686708.aspx*.

[MPA05] J. Manson, W. Pugh, and S. Adve. "The Java memory model." In *Proceedings of POPL*, pp. 378–391. ACM, 2005.

[MSM09] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. "DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages." Technical Report 090021, UCLA Computer Science Department, November 2009.

[MSM10] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. "DRFx: A simple and efficient memory model for concurrent programming languages." In *PLDI '10*, 2010.

[MSQ09] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. "SigRace: signature-based data race detection." In *ISCA*, 2009.

[NAW06] M. Naik, A. Aiken, and J. Whaley. "Effective static race detection for Java." In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pp. 308–319, 2006.

[Net93] R. H. B. Netzer. "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs." In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 1–11, 1993.

[NG09] Vijay Nagarajan and Rajiv Gupta. "Speculative Optimizations for Parallel Programs on Multicores." In *LCPC*, pp. 323–337, 2009.

[Nis04] H. Nishiyama. "Detecting Data Races using Dynamic Escape Analysis based on Read Barrier." *Third Virtual Machine Research & Technology Symposium*, pp. 127–138, May 2004.

[OC03] R. O'Callahan and J. D. Choi. "Hybrid dynamic data race detection." In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 167–178, 2003.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. "A better x86 memory model: x86-TSO." In *In TPHOLs '09: Conference on Theorem Proving in Higher Order Logics, volume 5674 of LNCS*, pp. 391–407. Springer, 2009.

[PFH06] P. Pratikakis, J. S. Foster, and M. Hicks. "LOCKSMITH: Context-sensitive correlation analysis for race detection." In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pp. 320–331, 2006.

[PG01] C. von Praun and T. R. Gross. "Object race detection." In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pp. 70–82, 2001.

[PGM00] Matt Postiff, David Greene, and Trevor N. Mudge. "The store-load address table and speculative register promotion." In *MICRO*, pp. 235–244, 2000.

[PK96] D. Perkovic and P. J. Keleher. "Online Data-Race Detection via Coherency Guarantees." In *OSDI*, pp. 47–57, 1996.

[Pou] Kevin Poulson. "Tracking the Blackout Bug." `http://www.securityfocus.com/news/8412`.

[PS03] E. Pozniansky and A. Schuster. "Efficient on-the-fly data race detection in multithreaded C++ programs." In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 179–190, 2003.

[PT03] 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.

[QW04] S. Qadeer and D. Wu. "KISS: Keep it simple and sequential." In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 14–24, 2004.

[RB00] M. Ronsse and K. de Bosschere. "Non-intrusive On-the-fly Data Race Detection using Execution Replay." In *Proceedings of Automated and Algorithmic Debugging*, Nov 2000.

[RL98] B. Richards and J. R. Larus. "Protocol based Data Race Detection." In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pp. 40–47. ACM Press, 1998.

[RPA97] P. Ranganathan, V.S. Pai, and S.V. 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*, pp. 199–210, 1997.

[SA08] Jaroslav Sevcík and David Aspinall. "On Validity of Program Transformations in the Java Memory Model." In *ECOOP*, pp. 27–51, 2008.

[SAW05] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. "Automated type-based analysis of data races and atomicity." In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 83–94, 2005.

[SBM06] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. "Accurate and efficient filtering for the Intel thread checker race detector." In *ASID '06: Proceedings of the 1st workshop on Architectural and system*

support for improving software dependability, pp. 34–41, New York, NY, USA, 2006. ACM.

[SBN97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs." *ACM Transactions on Computer Systems*, **15**(4):391–411, 1997.

[Sch89] E. Schonberg. "On-The-Fly Detection of Access Anomalies." In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[SFW05] Z. Sura, X. Fang, C.L. Wong, S.P. Midkiff, J. Lee, and D. Padua. "Compiler techniques for high performance sequentially consistent java programs." In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 2–13, 2005.

[SMN11] Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madanlal Musuvathi. "Efficient Processor Support for DRFx: Technical Report." Technical Report 110002, UCLA Computer Science Department, March 2011.

[SS88] 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.

[Ste93] N. Sterling. "WARLOCK - A Static Data Race Analysis Tool." In *Proceedings of the USENIX Winter Technical Conference*, pp. 97–106, 1993.

[Sut05] Herb Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software." *Dr. Dobb's Journal*, **30**(3):202–210, 2005.

[SVZ11] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "Relaxed-memory concurrency and verified compilation." In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pp. 43–54. ACM, 2011.

[TBB01] Walter Triebel, Joseph Bissell, and Rick Booth. *Programming Itaniumő-based Systems*. Intel Press, 2001.

[VJL07] J. W. Voung, R. Jhala, and S. Lerner. "RELAY: Static race detection on millions of lines of code." In *ESEC-FSE '07: Proceedings of the*

the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 205–214, 2007.

[Wol89] M. Wolfe. "More iteration space tiling." In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pp. 655–664, New York, NY, USA, 1989. ACM.

[WOT95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations." In *ISCA'95*, pp. 24–36, 1995.

[Yea02] K.C. Yeager. "The MIPS R10000 superscalar microprocessor." *Micro, IEEE*, **16**(2):28–41, 2002.

[YRC05] Y. Yu, T. Rodeheffer, and W. Chen. "RaceTrack: efficient detection of data race conditions via adaptive tracking." In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pp. 221–234, 2005.